FACULTY OF COMPUTING AND INFORMATICS

DEPARTMENT OF COMPUTER SCIENCE

Title

`Velo`:A UNIFYING DOMAIN-SPECIFIC LANGUAGE TO ABSTRACT CONTAINERIZATION AND ORCHESTRATION IN COMPLEX APPLICATIONS

Gervasius Ishuuwa

Submitted in fulfillment of the requirements for the degree of **Master** in **Computer Science**

Supervisor

Prof. José Quenum

DECLARATION

I, Gervasius Ishuuwa, hereby declare that the work contained in the report for the degree Masters of Computer Science, entitled: "`velo` - A unifying Domain-specific language to abstract containerization and orchestration in Complex Applications", is my own original work and that I have not previously in its entirety or in part submitted it at any University or other higher education institution for the award of a degree.

I further declare that I will fully acknowledge any sources of information I will use for the research in accordance with the Institution rules.

Signature _____ Date _____

RETENTION

Retention and Use of Thesis
I Gervasius Ishuuwa, being a candidate for the degree of Master of Computer Science accept the requirements of the Namibia University of Science and Technology relating to the retention and use of theses/mini theses deposited in the Library and Information Services.

In terms of these conditions, I agree that the original of my thesis/mini thesis deposited in the Library and Information Services will be accessible for purposes of study and research, in accordance with the normal conditions established by the Librarian for the care, loan or reproduction of theses/mini theses.

Signature _____ Date _____

METADATA

TITLE:	Mr
STUDENT NAME:	Gervasius Ishuuwa
SUPERVISOR:	Prof. José Quenum
DEPARTMENT:	Computer Science
QUALIFICATION:	Master of Computer Science
SPECIALISATION:	Computer Science
STUDY TITLE:	`velo` – A Unifying Domain-Specific Language to Abstract Containerization and Orchestration in Complex Applications
MAIN KNOWLEDGE AREA:	Application Containerization and Orchestration
KEYWORDS:	Containerization · Orchestration · Virtualisation · Domain-specific Language · Source-to-Source Compiler
TYPE OF RESEARCH:	Experimental Research
METHODOLOGY:	Quantitative Research Design With An Experimental Approach
STATUS:	Research Thesis
DOCUMENT DATE:	February 10, 2020
SPONSOR: (or Cluster/ Research lab)	NUST

Abstract

Application containerisation has been introduced to mitigate the discrepancies of the execution environment configuration and set up a complex application goes through from development to production, including other steps such as testing, staging, etc. Through an isolation mechanism, containers bundle into one package (binaries plus all their dependencies) the entire runtime environment required for an application or any of its components. Depending on the complexity of the adopted architectural style, container orchestration, the management and provisioning of containers, their load-balancing, security, scaling, and their network configuration might be needed to smoothen the overall deployment and execution experience. In the peculiar case of cloud-native applications, containerisation and orchestration are even more crucial.

Application containerisation and orchestration have risen as two interrelated technologies. However, they are handled with different toolset and formalisms. This increases the complexity of deploying such applications due to various moving parts. Moreover, for an independent team to try out some components of an existing application, they now have to stick to the prior deployment decisions made on their behalf. We argue that a better approach is to describe the desired state of both containerisation and orchestration and leave it to each team to decide on the actual tools and infrastructure they intend to use.

In this research, we introduce `velo`, a unifying abstraction domain-specific language (DSL) for application containerisation and orchestration. Intuitively, `velo` lets the user express the *desired* state of containerisation and orchestration for a complex application. It has two components: (1) an abstract specification language that describes the containerisation and orchestration for a complex application; and (2) a transpiler, a source-to-source compiler into both a container-specific and an orchestration-specific environments.

In order to define the specification language, we studied various containerisation and orchestration tools, including `docker`, `rocket`, `kubernetes`, `mesos marathon`, `docker compose` and `docker swarm`. The resulting concepts are centred around a *virtual bag*, a space within the infrastructure where containers can be run, and a *container*, an isolated space in the cluster where various processes corresponding to services will run and access resources (CPU, network, I/O, etc.). Containers are instantiated within a virtual bag. Each of these concepts can be further refined and represented in different ways. As well, several fine-grained descriptors are introduced to complete the specification of an application containerisation and orchestration. Furthermore, these descriptors can be provided during specification time or at compilation.

The compiler in `velo` is implemented against the grammar defined for

the specification language. It gives the user the possibility to generate both the container file and the orchestration description based on the same initial specification. Currently, we generate a `Dockerfile` for the containerisation, and `kubernetes`, `mesos marathon` and `docker compose` for orchestration. Following the grammar, the compiler can also automatically detect whether or not to prompt the user for missing descriptors.

We conducted a theoretical and practical evaluation of `velo`. The theoretical evaluation focuses on the semantics of the specification language as well as the correctness of its compiler. As for the practical evaluation, we tested `velo` following various scenarios and discuss our findings.

Keywords

Containerization · Orchestration · Virtualisation · Domain-specific Language · Source-to-Source Compiler

Contents

List of Figures

List of Tables

ACKNOWLEDGEMENT

Firstly, I would like to express my sincere gratitude to my supervisor Prof. José Quenum for the continuous support of my research, for his patience, motivation, and immense knowledge. His guidance helped me all the time of research and writing of this thesis. I could not have imagined having a better supervisor and mentor for my research.

I must express my very profound gratitude to Heinrich Aluvilu, Jonas Tomanga and Lameck Amugogo for providing me with unfailing support and continuous encouragement throughout the process of researching and writing this thesis. This research would not have been possible without them. Thank you.

Finally, I thank existence for this moment.

Chapter 1

Introduction

1.1 Background

The principle of *least privilege*, which requires that each program and every user of the system should operate using the least set of privileges necessary to complete a job, is one of the core principles in operating systems (OS) (Saltzer and Schroeder, 1975). However, many popular OS, e.g., Unix, Linux, etc. do not strongly implement that principle. For example, in Unix, modules such as the file system, the network stack, etc. are globally visible to all users. This makes it challenging to host independent applications which require different versions of the same library. For example, hosting two independent applications `App1` and `App2`, each requiring a different version of Python might prove challenging.

To address this limitation, system administrators generally deploy each application on a separate machine, isolating the conflicting libraries. More and more of these machines are virtual machines (VM). While VMs are very good at isolation, they require the emulation of the entire machine; all resources (CPU, memory, IO devices) must be virtualized.

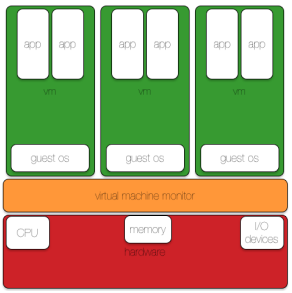

Figure 1.1: Full Virtualisation

Although virtual machines can be used to address the need for application isolation, they prove to be an inefficient choice when used only for that

purpose (Kratzke, 2017a; Soltesz et al., 2007; Felter et al., 2015). Generally, virtual machines experience a high warm-up time and inefficient use of system resources. Figure 1.1 depicts an overall architecture of full virtualisation with a hypervisor (also called virtual machine monitor) and virtual machines.

Application containerisation, on the other hand, proves to be a lighter and more efficient approach to addressing the application isolation need in a cloud environment. Put another way, the approach consists of running applications on an OS such that each application is isolated from the rest of the system; virtualise the OS to all applications instead of virtualising hardware to virtual machines. Figure 1.2 depicts an example of application containerisation using `Docker` (doc,) as the container runtime. Each container isolates the application that it is responsible for and the required binaries and libraries.

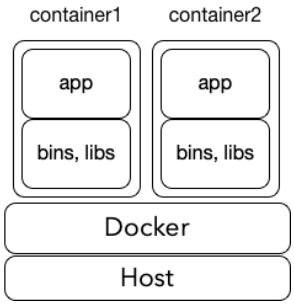

Figure 1.2: Application Containerisation

A modern container is more than just an isolation mechanism. Similar to how shipping containers are today, application containers are a unit of software that behaves in the same way everywhere regardless of what code and dependencies are included. Application containers enable developers to transport complex applications across infrastructures and various environments without requiring any modifications and regardless of the varying configurations in the different environments.

However, with complex applications, seldom does a production environment amounts to a single node. Thus, the containers need to be orchestrated: provisioning and management of containers, handling their scalability, security, load-balancing, networking, etc. A new set of tools is then introduced for that purpose. Examples of orchestrator include `Kubernetes` (kub,).

A careful observation then comes as follows. From initially being touted as a technology with many advantage points, including flexibility and ease of use, deploying a complex modular application (e.g., following the microservices architectural style), turns out to be a complex task. In other terms, the increase in flexibility of application development and deployment using microservice and containerisation and orchestration is matched with the in-

crease in complexity through many moving parts. In addition to mastering the development environment required for such applications, a developer also needs to understand application containerisation and orchestration. What exacerbates the issue is that although containerisation and orchestration are somehow inter-related technologies, they involve different tools and different independent description or specification languages.

In this research, we present `velo`, a unifying domain-specific language (DSL) that abstracts containerisation and orchestration in a complex application deployment. First, we identify the concepts of application containerisation and orchestration and assemble them into a DSL. The latter allows a developer to express a desired state for the application deployment. Once a valid specification is obtained, it is submitted to a source-to-source compiler, which will then generate the corresponding configuration for specific container runtime and orchestration tools.

1.2 Thesis Outline

This thesis consists of seven (7) chapters:

Chapter 1 Introduction discusses the background to our work;

Chapter 2 Fundamentals and Key Concepts provides an overview of the fundamental concepts needed as a foundation for the thesis;

Chapter 3 The Research Problem states the problem being addressed and discusses the research questions, objectives, limitations, benefits and expected outcomes of the work;

Chapter 4 Literature Review presents a literature review of research work similar to `Velo` followed by a comparison grid;

Chapter 5 `velo` – Language and Compiler details the domain-specific language we introduced and the approach we followed to build the transpiler;

Chapter 6 Evaluation presents the experiments conducted with `Velo` and discusses our evaluation of the transpiler;

Chapter 7 Conclusion draws important conclusions and sheds light on future work.

Chapter 2

Fundamentals and Key Concepts

This chapter provides an overview of the fundamental concepts needed to build a foundation for the thesis. These are key to understanding the concepts introduced in `velo`.

2.1 Containerisation

Containerisation is a product of operating system virtualization, instead of emulating the hardware, several user spaces are created under the same operating system kernel, and the hardware is used natively. A container provides a generic way of isolating a set of processes and resources such as memory, CPU, disk, etc., from the host. As such, no process within a specific user space can interfere with any other process within another userspace. In short, a container can be perceived as a lightweight operating system running inside the host system, natively executing program instruction (Dua et al., 2014).

The following sub-sections describe the first set of building blocks responsible for the management of a container on a single host. These building blocks are common among container systems, but each system provides its specific implementation.

2.1.1 Runtime Engine

A (Container) Runtime Engine provides the interface for life cycle management of single containers. Thus, a runtime engine spawns, starts, stops and destroys a container. Runtime engines also provide snapshot and clone functionality for a container and control resource consumption and container access to a host system resource. The runtime receives the container image

(fully described in section 2.1.3) as input and its responsible for unpacking, interpreting and executing the image.

2.1.2 Driver

The Runtime Engines relies on the container Driver to map life cycle controls and resource control to lower-level system calls. Hence, a Driver is responsible for the execution of kernel-specific functions. In Linux for example, this comes down to namespaces and cgroups. There are various container drivers. The default one of `Docker` is **runC** (run,), which was recently donated to the Open Container Initiative (OCI), but, started with their proprietary driver, called libcontainer. Others include **LXC** (lxc,), **OpenVZ** (ope,), **FreeBSD Jails** (bsd,) and **Solaris Zones** (sol,).

2.1.3 Container Image and Image Description

An Image Description is a file containing command-like descriptions for the creation of an Image. The result of the Image description when executed is the Image; the binary file containing all the libraries and user changes on top of the very basic operating system kernel. In the case of Docker, for example, as shown in Figure 2.1, Images are created with the build command. In turn, Images produce a Container (instance of an Image) when started with the run command.

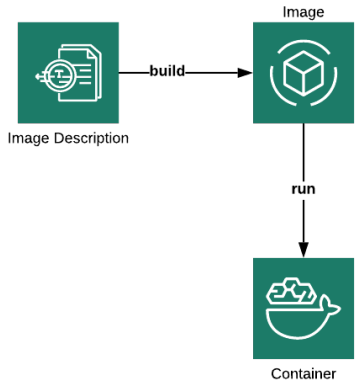

Figure 2.1: Container Image and Image Description

2.1.4 Registry

A registry is a repository of images, where users can upload their created images or can download a pre-built image. A registry can be public or private.

2.1.5 Examples of Container System

In this section, we discuss the two most popular container, systems namely, Docker and Rkt.

Docker

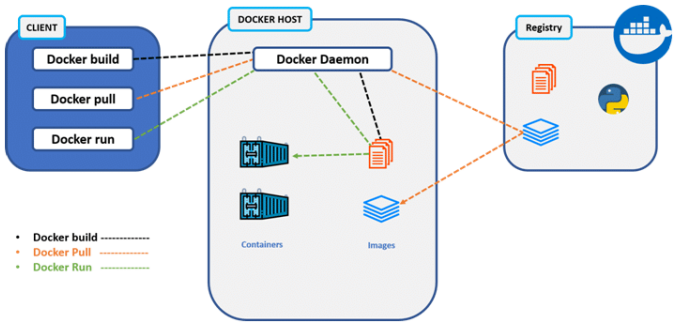

Figure 2.2: Docker Architecture

Docker is described as an open-source engine that automates the deployment of any application as a lightweight, portable, self-sufficient container that will run virtually anywhere. (doc,).

Docker makes it easier for organizations to automate infrastructure, isolate applications, maintain consistency and improve resource utilization. Docker uses the components depicted in Figure 2.2 to manage containers as follows:

Docker Client, a command-line client binary sends requests to the *daemon* also known as *dockerd* or *Docker Engine* through a RESTful API. The *daemon* is responsible for processing these requests and manage the containers accordingly.

To create a *container*, an *image* must first be built by running the `docker build` command. The result is a binary file that contains the installation steps of an application instance with its software and dependencies. Docker images can be built from a *Dockerfile*.

The *Dockerfile* is the core component of the portability of Docker. This file contains the instructions on how to build the Docker Image. The *Dockerfile* is highly portable and can be shared, versioned and stored.

Rkt

rkt (rkt,) is a container engine for Linux developed by CoreOS [1]. It is an implementation of App Container (APPC) [2], which is open source and defines

[1] `https://coreos.com/`
[2] `https://github.com/appc/spec/`

an image format, the runtime environment, and the discovery mechanism of application containers.

rkt supports App Container Image (ACI) [3] as well as Docker images via the docker2aci[4]. A pod (the basic unit of execution in rkt) is a grouping of one or more images (ACIs), with some optionally applied additional metadata on the pod levelfor example, applying some resource constraints, such as CPU usage.

From the start, it has been designed to be composable and secure. *rkt* is composable in that it makes use of existing technologies systemd, gpg, tar, http, etc. Moreover, one can choose various isolation environments; container: uses systemd-nspawn to execute containers, VM (stage1-kvm): runs a container within a virtual machine with its operating system kernel and hypervisor (KVM) isolation, rather than creating a container using Linux cgroups and namespaces, or none (stage1-fly): run a container using only chroot isolation.

rkt was built with security in mind from the beginning with image signing and verification. The `insecure-options` flag is required for insecure actions.

Because *rkt* employs swappable isolation mechanisms, depicted in Figure 2.3 execution in *rkt* is divided into several distinct stages.

The first stage, *stage0* is responsible for image discovery and retrieval and sets up a filesystem for stages 1 and 2.

The next stage *stage1* sets up the execution environment for the container execution using the filesystem set up by stage0. *rkt* uses systemd-nspawn to set up cgroups and networking in this stage. The goal here is to keep *stage1* swappable by other implementations.

The final stage, *stage2*, is the actual execution of the pod using the execution environment set up by *stage1* and filesystem set up by *stage0*.

[3] `https://github.com/appc/spec/blob/master/spec/aci.md`
[4] `https://github.com/appc/docker2aci`

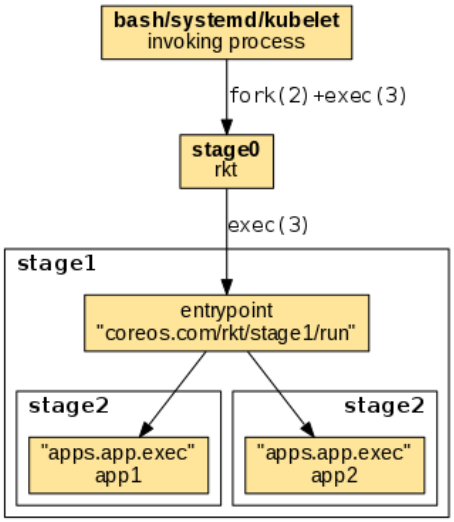

Figure 2.3: rkt Execution Flow

2.2 Comparison of container, virtual machine and unikernel

Unikernels are a relatively new concept in which software is directly integrated with the kernel it is running on. This happens by compiling source code, along with only the required system calls and drivers, into one executable program using a single address space (Pavlicek, 2016).

Although they both aim to make computing more responsive and less resource-intensive, the natures of unikernels and containers contrast significantly. The big difference is that containers require and depend on a normal OS, and unikernels do not. Instead, there is only what is necessary to achieve a specific function, namely application code, along with the minimum necessary OS functionality to run the application. This grouping of code and supporting OS components lives independently.

Figure 2.4 illustrates this concept by comparing virtual machines, containers and unikernels. The figure uses a blue color to indicate hardware or hypervisors, orange to indicate kernel space and green to indicate user space.

The claimed advantages of unikernels over containers and virtual machines generally include security and performance (Madhavapeddy and Scott, 2013). The reduced kernel and system complexity can make a unikernel much faster than a regular virtual machine (Briggs et al., 2014). Security is improved by the reduced attack surface, because of the way unikernels are built , they are less vulnerable to security problems, since there are typically fewer attack options, except the program, the required libraries and

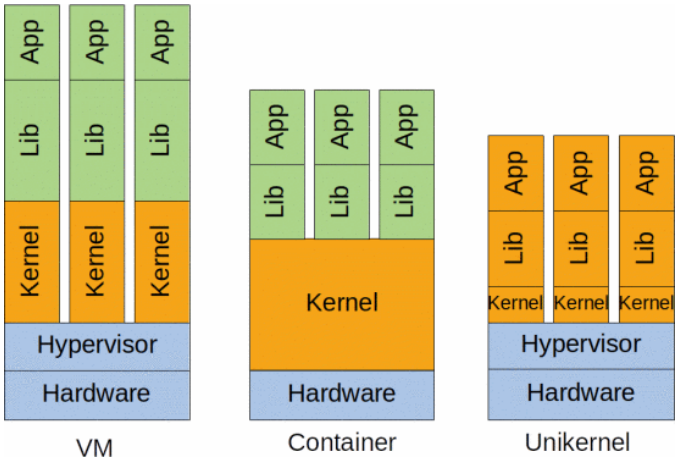

Figure 2.4: Comparison of virtual machine, container and unikernel system architecture

the kernel functions it uses.

The downside is that programs always run in kernel mode, making it easier for bugs and hacks that do succeed to critically break the machine, while making it harder to debug (Pavlicek, 2016) as unikernels usually do not have their own sets of debugging tools and existing ones would have to be cross-compiled.

Although virtual machines can be used to address the need for application isolation, they have some essential differences from containers. Virtual machines use hypervisors to emulate the hardware (CPU, memory and I/O devices). Instead of emulating the hardware, containers create several user spaces on the same OS kernel, and the hardware is used natively. Virtual machines provide complete isolation from the host operating system and other VMs. Containers typically provide lightweight isolation from the host and other containers.

2.3 Orchestration

Orchestration is the set of operations that either manually or automatically, are responsible for selecting, deploying, monitoring, and dynamically controlling the configuration of containers in a cluster to guarantee a certain level of quality of service. (Di Martino et al., 2015). The following subsections describe the second set of building blocks that are responsible for the management of containers on distributed hosts. These building blocks are common among orchestrators, but each orchestrator provides its specific implementation.

Coherent unit	Docker Swarm	Kubernetes	Marathon
Pod		Pod	Pod
Service	Service	Service + Deployment	Pod
Deployment	Service	Deployment	Pod
ReplicaSet		ReplicaSet	
SatefulSet		SatefulSet	
DaemonSet		DaemonSet	

Table 2.1: Concepts mapping

2.3.1 Orchestrator

An *orchestrator* manages deployment procedures and environmental configurations of containers across multiple servers. To accomplish the management of different types of containers with inter-dependencies and specific requirements, an orchestrator relies on the concept of *coherent units* which in `velo` translates to *virtual bag*. This concept plays a vital role in managing containers across a cluster: A coherent unit logically combines a set of containers, which will be managed as one. Containers which are part of the same coherent unit share resources and a common life cycle.

We have identified *coherent units* which we found to be distinguishable when using orchestrators. These are:

Pod an encapsulation of an application container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should be run,

Service an abstraction which defines a logical set of *Pods* and a policy by which to access them,

Deployment provides declarative updates for *Pod*

ReplicaSets ensures that a specified number of *Pod* replicas are running at any given time,

SatefulSet manages *Pods* that are based on an identical container spec. Maintaining a persistent identifier across any rescheduling,

DaemonSet ensures that all (or some) nodes in the cluster run a copy of a *Pod*.

Table 2.1 shows how these *coherent units* are represented in different orchestrators.

2.3.2 Discovery Service

A Discovery Service exposes static endpoints which are dynamically mapped to running containers. Each host runs a component, called discovery agent, which monitors its local containers life cycles and publishes the state of each container. Existing orchestrators either include discovery agents or can be integrated with an external discovery service.

2.3.3 Overlay Networks

Overlay networks are software solutions, which run agents on each host. The overlay network handles the mapping of network ports and traffic. Moreover, they map logical network interfaces to physical network interfaces to enable communication between containers on the same host as well as across multiple hosts.

2.3.4 Scheduler

A scheduler receives the specification for a container or a coherent unit of multiple containers and manages life cycle and placement over multiple hosts. Schedulers ensure that resource capacity constraints are not violated and handle load distribution over a cluster of machines. Schedulers consequently manage resource allocation of containers.

2.3.5 State Storage

Features, such as the already discussed scheduling and discovery, require a consistent and up-to-date view of the cluster state. This state is stored in a database service, which consequently becomes an integral part of the container ecosystem. Due to the potentially high number of containers which are spread across multiple hosts, the database should be fast, natively distributed and scale well with the number of requests.

The data model for the state is kept rather simple; key-value stores dominate the currently existing solutions. Current systems also share a subscription/listening capability, which allows clients to be automatically informed about updates. All popular projects rely on consensus algorithms to ensure some form of consistency (Netto et al., 2017). Apache Zookeeper (Hunt et al., 2010) is a well-known system fulfilling these requirements, implementing the ZAB protocol (Junqueira et al., 2011) for consensus. Other systems are etcd[5] and Consul[6], which both use the Raft (Ongaro and Ousterhout, 2014) consensus protocol.

[5] `https://etcd.io/`
[6] `https://www.consul.io/`

2.3.6 Resource Monitor

Resource Monitors collect resource utilisation and health data on per container, host and cluster basis. In a virtual machine, it is possible to use the operating system as a means to measure resource consumption, but for a container, this is not possible, as, each container sees all host resources. As a result, measures like CPU utilisation are invalid when measured from inside a container as they reflect the overall state of the host system.

As a solution orchestrators run monitoring systems which integrate with the containers driver's local resource allocation. Dockers *stats* command provides resource metrics data either in textual or JSON format. cAdvisor[7] is an example of a per container-based solution with pluggable driver support. Whereas, Metrics Server[8] is a cluster-wide aggregator of resource usage data.

2.3.7 Examples of Orchestration System

In this section, we discuss the most popular orchestration systems, namely, Kubernetes, Docker Swarm and Mesos Marathon.

Kubernetes

Kubernetes (kub,) is an open-source platform for automating container operations such as deployment, scheduling and scalability across a cluster of nodes. Google has developed it. Furthermore, it is a similar approach to Borg and Omega (Burns et al., 2016), those projects were used within Google for a long time and considered as a first unified container management system developed in Google but not considered as open-source.

Kubernetes allows the user to declaratively specify the desired state of a cluster using high-level primitives in `YAML`. For example, the user may specify that they want three instances of an application container running. Kubernetes self-healing mechanisms, such as auto-restarting, rescheduling and replicating containers, together, work on the actual state to achieve the desired state.

Figure 2.5 depicts a typical `Kubernetes` architecture. It shows a *master server*, which controls the overall cluster through an *API server*, a *controller manager*, a *scheduler* and an instance of `etcd`. The second type of component is a *node server*, which contains a *kubelet*, a contact point of each node within the cluster, a *kube-proxy*, handling the networking, a container runtime and a collection of pods where the actual containers are run.

[7] `https://github.com/google/cadvisor`
[8] `https://github.com/kubernetes-incubator/metrics-server`

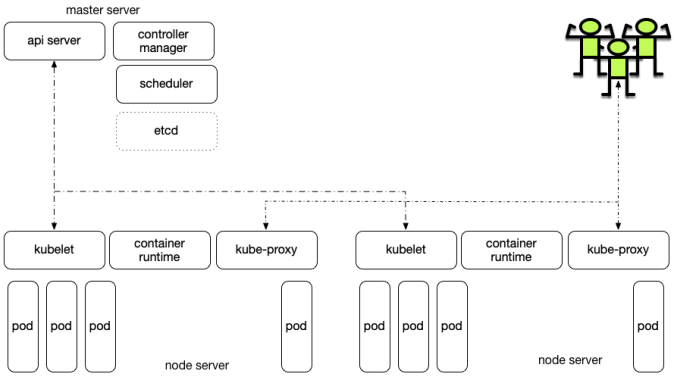

Figure 2.5: Kubernetes Architecture

Docker Compose and Docker Swarm

Compose[9] is a tool for defining and running multi-container Docker applications. *Compose* uses a `YAML` file to configure the applications services. Then, with a single command, all the services from the configuration are created and started.

Although *Compose* enables multi-container applications, they are still restricted to a single host. *Docker Swarm*[10] then complements *Compose* by allowing the creation of a cluster of Docker Engines as depicted in Figure 2.6.

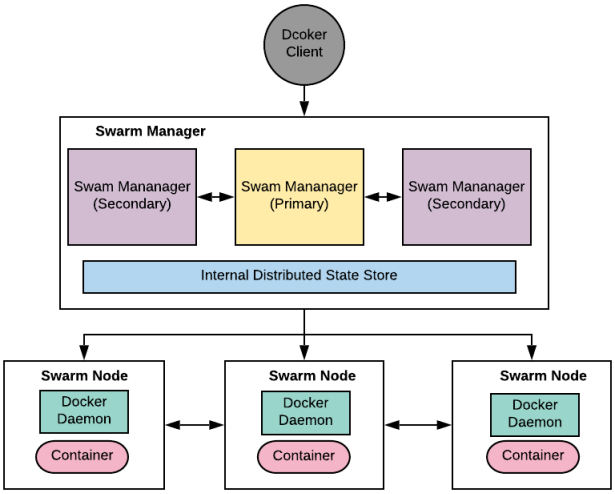

Figure 2.6: Docker Swarm Architecture

[9] `https://docs.docker.com/compose/`
[10] `https://docs.docker.com/engine/swarm/`

Mesos Marathon

Marathon [11] is a production-grade container orchestration platform for Mesospheres Datacenter Operating System (DC\OS) and Apache Mesos. *Marathon* provides a RESTful API for starting, stopping, and scaling applications. *Marathon* has native support for both Mesos containers (using cgroups) and Docker containers.

Applications are an integral concept in *Marathon*. Each *application* typically represents a long-running service, of which there would be many instances running on multiple hosts. An *application* instance is called a *task*. The application definition described in `JSON` specifies everything needed to start and maintain the tasks.

Figure 2.7 graphically shows how *Marathon* handles deployments to Mesos. *Marathon* uses *Zookeeper* to synchronise the configuration of the cluster so that it has redundancy and can recover if one of the hosts in the cluster goes down. *Marathon* connects to the slaves where the tasks run.

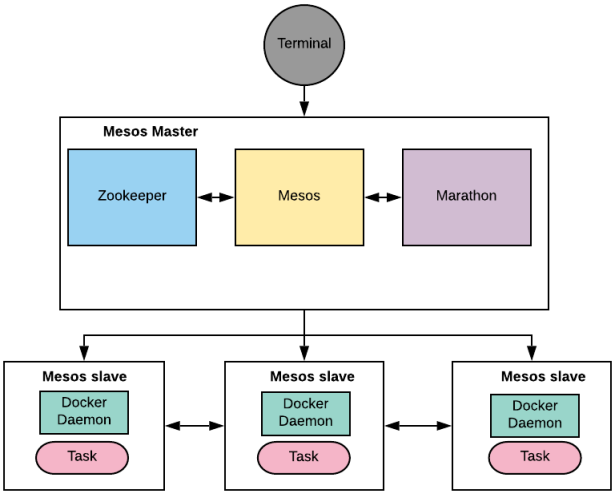

Figure 2.7: Marathon Architecture

[11] `https://mesosphere.github.io/marathon/`

Chapter 3

The Research Problem

3.1 Problem Statement

As the adoption criteria of *cloud computing* get better understood, and its benefits clearer (e.g., cost reduction, reliability, flexibility and maintenance), more and more organisations and customers are adopting it (Oliveira et al., 2014; Hassan, 2017). Cloud computing allows an organisation to run its entire IT infrastructure through shared physical storage and networking capabilities. Equally, it enables a team of developers to gain access to a platform where they can deploy their applications without having to worry about the underlying operating system and network functionality. The flexibility gained with cloud computing has prompted software developers to move away from monolithic applications and embrace the microservice architectural style (Dragoni et al., 2016; Dragoni et al., 2017; Molnár et al., 2014). The latter, however, increases the need for application isolation.

Several application isolation techniques have been developed in the past, including the use of bare metal-based virtual machines (de Alfonso et al., 2017) as well as private and public cloud infrastructure (Kratzke, 2017b). Recently, *containerisation* and *orchestration* prove to be a better option due to lower startup time and resource optimisation in the infrastructure. Thanks to their early success, an increasing number of containerisation and orchestration technologies (specification formalisms, execution runtime, etc.) are being introduced (e.g., `Docker`, `Rocket`, `Docker Swarm`, `Kubernetes` and `Mesos Marathon`) (de Alfonso et al., 2017; Kratzke, 2017a).

However, there are still various gaps in most containerization and orchestration platforms (Quint and Kratzke, 2018). Most of these technologies, although inter-related, lack interoperability. Furthermore, the increase in containerisation and orchestration specification formalisms limits the expected flexibility in complex and cloud-native application deployment. These issues are even more acute for multi-cloud contexts as well (Quint and Kratzke, 2018). In this section, we highlight the issues hindering containerisation and

orchestration specification.

3.1.1 Specification Formalisms

The success of most orchestration and containerisation technologies stems from their declarative approach. The desired state of the system in terms of orchestration and containerisation is specified in configuration files. However, the lack of high-level abstraction as well as unifying underlying concepts for both containerisation and orchestration hinders complex application containers and their orchestration. By way of illustration, consider a software development team using different container and orchestration technologies between the testing, staging and production phases of application development. Specifying the desired state that can adequately be translated into specific technologies for each phase is currently not feasible. Moreover, the possibility for different development teams using a different toolset to deploy an application using the same application image and image description cannot be achieved without a fair amount of manual configurations.

3.1.2 Portability and Inter-operability

New trends in software development are to write code that can be shipped instantly (Rodrguez et al., 2017). Development teams wish to continuously build and deploy their software and drastically reduce the release cycle. Through automated deployment, *containers* enable developers to focus on the design and testing aspects of the software development process. However, various container management tools (with different standards) end up being used within an organisation. This exacerbates the *portability* and *interoperability* issues. In short, in order to migrate a Docker Compose workflow to Kubernetes for example, one needs to translate Compose service definitions to Kubernetes objects, manually. The scheduling of microservices to the cloud is tied to interoperability, application topology and composition aspects as well as elastic runtime adaptation aspects of cloud-native applications (Fazio et al., 2016; Saatkamp et al., 2017). To the best of our knowledge, the combination of these two aspects (interoperability and application topology definition/composition in heterogeneous environments) has not been solved satisfactorily so far.

3.1.3 Enforcing Best Practices

While the number of containerisation and orchestration specification formalism has been growing steadily, there is no standard to constrain the users with the concept to use within a given context. Generally, new versions of a formalism include a set of *best practices*. However, there is no mechanism to enforce those best practices.

3.1.4 Towards a Unifying Abstraction

We argue that (substantial) variations in the inner workings of the emerging containerisation and orchestration technologies do not pose a challenge. Instead, we perceive the real challenge in keeping these specifications as similar as possible or inter-operable. With different types of container tools diverging from one another, comes the looming effect of proprietary standards which introduces the risk of vendor lock-in (Linthicum, 2016). As such, the promise containers have been touted for, i.e. provide a common abstraction layer that allows applications to be localized within the container, and then ported to other computing environments, is threatened by these competing standards and formats (Linthicum, 2016).

In this research, we introduce a domain-specific language (DSL) to unify the description of application containers and their orchestration. Using the new language, the container requirements can be described abstractly, ignoring specific execution details. The language is accompanied by a transpiler, which can then be used to generate specific descriptions for a concrete target container and orchestration technology. Using this approach, several developer teams or users can share the same container description while adopting different container technologies without any need to tweak any code or configuration.

3.2 Research Objectives and Questions

Our main objective is to design a DSL to abstract and unify container and orchestration specification and build a transpiler that will translate the generic description to specific ones. To this end, four specific sub-objectives were pursued.

3.2.1 Research Objectives

- Analyse the scope and functionality of popular containerisation and orchestration tools;
- Design a DSL to unify container descriptions and their orchestration;
- Develop a transpiler to translate the DSL to a specific container and orchestration descriptions;
- Test and evaluate the DSL and the transpiler.

3.2.2 Research Questions

The questions that will guide our research come as follows:

1. What is the scope and functionality of existing/popular containerisation and orchestration tools?
2. Through what means can one unify the various functionality and standards?
3. How can the translation of a generic description into specific descriptions be automated?

3.2.3 Limitations

One of the obvious limitations that this study experienced is that the fact that containerization and orchestration are relatively new technologies there is a limited amount of research work in the DSL field for containerisation and orchestration.

3.2.4 Expected Outcomes

The main outcome of this research is a prototype of a DSL and a transpiler. In this prototype, both containerisation and orchestration specifications are combined in an abstract description. The latter is then passed to the transpiler to separately generate a specific description for container technology and an orchestration tool. For containerisation we focus on `Docker`, while we offer `kubernetes`, `Docker Compose` and `Mesos Marathon` for orchestration.

3.2.5 Significance/Contribution

The contributions of this research are manifold. At a technical level, it will increase the productivity of software development teams using heterogeneous containerisation and orchestration tools. At a conceptual and methodological level, this work will advance the efforts in process automation in complex application development. These efforts include the DevOps movement, efforts on continuous integration and deployment, etc. Finally, this research will offer a better perspective towards unifying containerisation and orchestration and pave the way to standardisation.

Chapter 4

Literature Review

4.1 Standardisation Efforts

In contrast to calls for consolidation (Peinl and Holzschuher, 2015), the container ecosystem is still growing both in existing tool categories as well as in new ones. Rather than allowing fragmentation to impact the momentum of container technologies, the IT industry came together with intending to set standards implemented as open-source projects.

The Open Container Initiative (OCI) governed by the Linux Foundation, was started recently to help address the issue of fragmentation in the world of application containers. Docker and CoreOS have decided to cooperate to define specifications around containers. The OCI was designed to collaborate on an open standard container image format and runtime specification, which will allow a compliant container to be portable across compliant operating systems and platforms.

While the industry has come together to support the OCI runtime, the critical aspect of image distribution is not currently addressed by the OCI. Consequently, this hinders end-user portability.

4.2 Domain-Specific Languages

Cloud computing appeared as unquestionable IT phenomena in recent times. Nowadays, applications are developed focusing on cloud (Binz et al., 2014), that means applications are developed as cloud-native applications, or the existing applications are moved into the cloud (Wilder, 2012).

Goncalves et al. present the first DSL explicitly designed to describe cloud entities, CloudML (Goncalves et al., 2011). CloudML is underpinned in the D-Cloud context aiming at allowing cloud computing providers to describe both cloud resources and services, and cloud developers, to describe their computing requirements. D-Clouds stands for Distributed Clouds, and the authors define it as smaller data centres sharing resources across geo-

graphic boundaries. CloudML is an XML-based language, and it is based on three requirements: (i) representation of physical and virtual resources as well as their state; (ii) representation of services provided; and (iii) representation of developers requirements.

Ferry et al. introduced CloudMF (Ferry et al., 2013). It aims at supporting provisioning and deployment of applications in multiple clouds runtime and design time. To accomplish this objective, CloudMF covers four requirements: (i) separation of concerns; (ii) provider independence; (iii) reusability; and (iv) abstraction. CloudMF consists of two components: (i) CloudML, the modelling environment DSL, and (ii) Models@run-time, which provides an abstract representation of the running system.

To enable the creation of portable cloud applications and the automation of their deployment and management, Topology and Orchestration Specification for Cloud Applications (TOCSA) (toc,) models the applications components, their relations, and management in a portable, standardized, machine-readable format. TOCSA is an Organization for the Advancement of Structured Information Standards (OASIS) standard that provides a YAML-based modelling language for specifying portable cloud applications, and for automating their deployment and management. It was introduced to enable the portability of applications between clouds and remedy vendor lock-in for its users (Binz et al., 2014). TOCSA permits describing the structure of a cloud application as a typed, directed topology graph, whose nodes represent application components, and whose arcs represent dependencies among such components. Each node of topology can also be associated with the corresponding components requirements, the operations to manage it, the capabilities it features, and the policies applied to it.

TOCSA is similar to `velo` in the sense that they are both DSLs, but, TOCSA is more a standard, e.g. applications described using TOCSA can only run on a TOCSA enabled cloud platform as depicted in Figure 4.1. Whereas `Velo`'s efforts are centred towards the unification of descriptions through abstraction, that is, Velo's output can be run on any platform running the supported container tools as depicted in Figure 4.2.

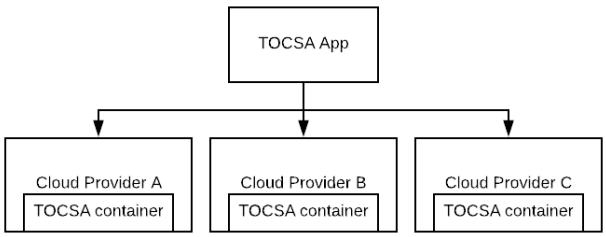

Figure 4.1: TOSCA Application

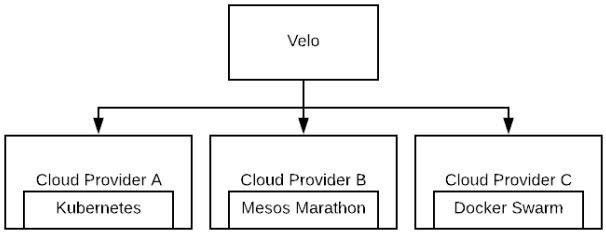

Figure 4.2: Velo Application

TOCSA uses Relationship Templates to represent the relations among the components, e.g., a "communicates with" relationship is used to define that a front end component communicates with a back end component. In contrast, `velo` uses label selectors that are used to create a service at runtime.

Because TOCSA is a standard, it does not provide any software. It, therefore, requires an ecosystem (Binz et al., 2014). In (Binz et al., 2013) Tobias et al. develop an open-source runtime environment for the TOCSA description, which is responsible for deployment and management. It processes the Cloud Service Archives (CSARs), runs plans and also manages the deployment states. The OpenTOSCA-Container handles the deployment and management of TOCSA compliant applications.

On the other hand, Cloudify (clo,) is an open source cloud orchestration tool developed by GigaSpaces Technologies. Cloudify is designed in such a way that it can support any application regardless of its stack (i.e. languages and dependencies). It can be deployed on any IaaS cloud, and provide full control over the underlying infrastructure to its users, such as monitoring all aspects of the deployed application, detecting issues and failure, manually or automatically repairing them and handle ongoing maintenance tasks.

Cloudify uses a TOCSA YAML Simple Profile (Palma et al., 2016) based DSL to define the execution plans for the lifecycle of the application, including installation, startup, termination, orchestration and monitoring. A significant component of Cloudify is the Cloudify DSL parser, which intends to read and validate the TOCSA blueprints and using its workflow engine to map TOCSA operations to its specific plugins.

Cloudify does fully comply with the TOCSA standard as it does not support all the features like CSARs and the parser looks for namespaces specific to Cloudify, which may lead to vendor lock-in.

It is essential to realise that standards in the IT world often carry the reputation of being slow to evolve, overly complex or not grounded by implementation experience, and controlled by only a few members. In the past, VMware embarked on defining an Open Virtualization Format (OVF) in

an attempt to unify the diverse virtual machine types created by vendors. However, although VMware's efforts around a standardised specification for virtual machines were received favourably, for the most part, the initiative did not reach the level of success the company intended to initially. The result is a fundamental lack of portability, which has been a pain point for customers and has led to the creation of multiple islands of infrastructure (Maskasky, 2014).

jclouds-docker (jcl,) is an open-source library for Java and Clojure that provides abstractions modelled on Docker, allowing developers to create portable containerized applications. jclouds-docker uses the Docker remote API to access Docker operations such as creating, listing, inspecting and starting and stopping a container.

In contrast to `velo`, jclouds-docker uses a General Programming Language (GPL) to execute Docker APIs. `velo`, on the other hand, does not interact with the API. Instead, we generate the described application DockerFile. jclouds-docker is limited to one container tool, Docker, while `velo` supports most popular platforms Docker, Kubernetes and Mesos Marathon.

In (Peinl et al., 2016), Peinl et al. survey a variety of containerisation tools for services in the cloud, particularly for those services that are not cloud-aware or cloud-native. The work identifies and addresses some of the gaps in containerisation and orchestration of services in the cloud. Their work focused more on addressing the various requirements, including communication of containers across hosts and migration of containers to different hosts for multi-container applications.

Nardelli et al. (Nardelli et al., 2017) propose an approach based on integer linear programming for the elastic provisioning of virtual machines to host application containers. They factor in the heterogeneity of container requirements and virtual machine resources. The work focuses more on developing an efficient technique for orchestration. It shares some closeness with `kubernetes` in its various functions around container orchestration.

In (Buyya et al., 2018) Buyya et al. propose an architectural framework for the efficient orchestration of containers in cloud environments. Their approach is centred around computing resource (e.g. memory, CPU, disk, etc.) scheduling and rescheduling policies as well as auto-scaling algorithms that enable the creation of elastic virtual clusters. The proposed framework enables the sharing of a computing environment between differing client applications packaged in containers, including web services, offline analytic jobs, and back end pre-processing tasks. They argue that solving the problem of scheduling, rescheduling and scaling policies for shared container-based virtual clusters in clouds maximises resource utilization and minimises resource costs while meeting the quality of service (QoS) requirements of various applications running on the clusters. Their underlying mechanism is an extension of `kubernetes` scheduling techniques.

4.3 Multi-Cloud Operations

Quint and Kratzke (Quint and Kratzke, 2018) prototyped a DSL that enables to describe cloud-native applications being transferable at runtime without downtime. This work mainly focuses on the deployment and execution of cloud-native applications in a multi-cloud environment. The early results show a model that allows various services in different clouds to be moved around without interrupting the overall execution of the complex application.

On the other hand, Cito et al. (Cito et al., 2015) argue that deployment costs are generally not a tangible factor for cloud developers. They approached the deployment problem from a development perspective and proposed CostHat (Leitner et al., 2016), a what-if, cost analysis model. It is used to implement tooling that warns cloud developers directly in the Integrated Development Environment (IDE) about certain classes of potentially costly code changes, enabling developers to build applications that are cloud-native, as opposed to applications that are merely migrated to cloud infrastructure. Here, the focus is not really on the deployment mechanism, but on the optimisation of what is being deployed to be more cloud-aware.

The various pieces of work discussed in this section address a specific aspect of application (whether cloud-native or not) deployment in the cloud. More particularly, most of them touch on containerisation and orchestration. However, they do not address the challenge of the growing diversity of tooling and configuration and specification languages for these two interrelated concepts. Our approach in this research is to introduce an abstraction mechanism that does not prohibit the diversity of tooling but maintains a consistency around them.

4.4 Summary

In this section we summarize the related work with their shortcomings and their strong points.

Related work	Strengths	Weaknesses
OCI	The OCI standards cover two key components of the container ecosystem, the image format(image-spec) for containers and the runtime specifications (runtime-spec). The OCI image format describes the way a container image is laid out internally and what the various components are, whereas the runtime specification describes how the container is configured, executed and disposed.	The critical aspect of image distribution is not currently addressed.
CloudML	CloudML brings a common language that can be implemented by different Cloud equipment vendors in order to integrate their different solutions. Certainly, such integration cannot be made without some common protocol implemented by the vendors, but CloudML offers a common terrain for data representation that is a crucial step towards interoperability.	CloudML is just a language. It does not provide any software. It therefore requires an ecosystem, like cloudMF for enacting the provisioning, deployment and adaptation of these modelled applications.
TOCSA	Provides a YAML-based modelling language for specifying portable cloud applications, and for automating their deployment and management.	Applications described using TOCSA can only run on a TOCSA enabled cloud platforms.
jclouds-docker	Uses native Docker APIs to access Docker operations such as creating, listing, inspecting and starting and stopping a container.	jclouds-docker is limited to one container tool, Docker.

Table 4.1: Strength and weaknesses of Approaches

Chapter 5

`velo` – Language and Compiler

5.1 The Syntax

The complete `velo` grammar (see appendix A) was defined using `antlr` (Parr and Quong, 1995; Parr and Fisher, 2011). Here, we discuss a concise excerpt[1] of the grammar presented in BNF notation (McCracken and Reilly, ; Zaytsev, 2012). As highlighted in the Introduction (See chapter 1), the purpose of the language is to abstract concepts related to orchestration and containerisation in complex applications. The main idea is to specify the desired state of both the orchestration of the virtual infrastructure (cluster) hosting an application as well as the containerisation of the various services composing the application.

`velo` is designed to be used in two modes: *light* and *rich*. In a rich mode, all the parameters (health check, scheduling, storage, etc.) required to run the containerisation and orchestration of an application are provided in the specification file. In a light mode, these parameters are omitted from the specification file and re-introduced during compilation.

To capture orchestration in `velo`, we introduced the concept of a *virtual bag* (*vbag*), a space within the infrastructure where containers can be run. A virtual bag could translate to the notion of *pod* in `kubernetes`, specified as a *Pod* **kind** or as a *Deployment* and *Service* as prescribed by the best practices. We further distinguish between two types of virtual bags; a virtual bag that can host a single container called *s_vbag* and another one that can host multiple containers, called *m_vbag*. The rationale behind that distinction is that there might only be a need for a lightweight deployment. In that case, the orchestration still provides services such as scalability, monitoring, etc. The other central concept in our specification, *containerisation*, is captured

[1]The complete `antlr` grammar can be accessed at `http://196.216.167.206:3000/gervasius/velo/grammar` and is available in appendix A

through the concept of *container*. Conceptually, a container is an isolated space within a virtual bag, where various processes corresponding to services will run and access resources (CPU, network, I/O, etc.). Containers are instantiated within a virtual bag.

The following BNF production rules summarize the syntax of a virtual bag. The metadata of a virtual bag includes its *identifier*, the number of expected *instances*, the *network* endpoints it has access to, the *users* who can authenticate to it, and its *version* number. *Labels*, when they are used, provide identifiers for entity names in the setup.

⟨*velo-app*⟩ ::= ⟨*velo_metadata**⟩ ⟨*orchestration*⟩

⟨*velo_metadata**⟩ ::= ⟨*author*⟩ | ⟨*title*⟩

⟨*author*⟩ ::= 'author' ⟨*string*⟩

⟨*title*⟩ ::= 'title' ⟨*string*⟩

⟨*orchestration*⟩ ::= ⟨*vbag*⟩

⟨*vbag*⟩ ::= ⟨*s_vbag*⟩|⟨*m_vbag+*⟩

⟨*s_vbag*⟩ ::= ⟨*v_metadata**⟩ ⟨*labels?*⟩ ⟨*container*⟩

⟨*m_vbag*⟩ ::= ⟨*v_metadata**⟩ ⟨*labels?*⟩ ⟨*container+*⟩

⟨*v_metadata*⟩ ::= ⟨*id*⟩ | ⟨*inst*⟩ | ⟨*nets*⟩ | ⟨*auth*⟩ | ⟨*version*⟩

⟨*labels*⟩ ::= ⟨*label+*⟩

⟨*label*⟩ ::= ⟨*label_name*⟩ '=' ⟨*string*⟩

⟨*label_name*⟩ ::= ⟨*string*⟩

Finally, a container has its *metadata*, the *image* it should be built from, *labels*, *resources*, *network* ports and protocols and a logical *volume* for data access and persistence. Inside the image of a container, there may be labels and commands, the actual instructions to be executed inside the container.

⟨*container*⟩ ::= ⟨*c_metadata**⟩⟨*image*⟩⟨*labels?*⟩⟨*resources?*⟩⟨*network*⟩⟨*volume*⟩

⟨*c_metadata*⟩ ::= ⟨*name*⟩ | ⟨*policy*⟩ | ⟨*auth*⟩

⟨*policy*⟩ ::= 'restart' '=' ⟨*policy_val*⟩

⟨*policy_val*⟩ ::= 'always' | 'on-failure' | 'no' | 'unless-stopped'

⟨*image*⟩ ::= ⟨*labels?*⟩ ⟨*commands*⟩

By way of example, consider a Web application consisting of a user interface in `Python` served by `Nginx` Web server. Figure 5.1 depicts the

interconnection between the components of the application. The following explains the concepts used in `velo`'s grammar.

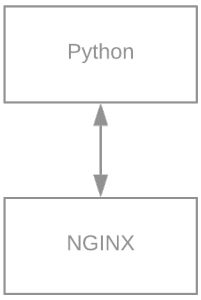

Figure 5.1: Web Application Deployment

The *mvBag* has *id*, which uniquely identifies the application; the expected number of instances of the application is denoted by *instances*. When there is a need for network communication between components of the application, the *networks* attribute provides a definition of all network names used in the infrastructure. Other attributes include *matchExpressions*, *user* and *mvBag.labels*.

matchExpressions is a grouping mechanism which groups containers according to the specified expression. *user* denotes the account from which the application will be running. *mvBag.labels* assign labels to the *mvBag* which can be used as an identity mechanism once the application is deployed to apply other services, such as restarting pods, who match these labels.

```
1  [mvBag]
2  id = "bookingwebapp"
3  instances = 2
4  networks = ["Net1","Net2"]
5  matchExpressions = [["tier","In","frontend"],[ "environment","NotIn","dev"]]
6  user = "gervasius"
7  version = "0.1"
8
9    [[mvBag.labels]]
10   App = "bookingwebapp"
11   Environment ="production"
```

Listing 5.1: mvBag specification

An *mvBag* can contain multiple containers denoted by *mvBag.containers*. A single *mvBag.containers* maps to the coherent unit introduced in section 2.1.

```
1  [[mvBag.containers]]
2  name = "pythonapp"
3  restart = "Always"
4  user = "admin"
```

Listing 5.2: mvBag.containers specification

Health checks may be specified per container to be run against that container, denoted by *mvBag.containers.healthcheck*.

```
[mvBag.containers.healthcheck]
gracePeriodSeconds = 30
ignoreHttp1xx = false
intervalSeconds = 30
maxConsecutiveFailures = 3
path = "pythonapp/bookings"
portIndex = 1
protocol = "TCP"
timeoutSeconds = 60
```

Listing 5.3: mvBag.containers.healthcheck specification

As depicted in Figure 2.1 a container is an instance of an Image, here denoted by *mvBag.containers.image*. An Image is built from a set of instructions denoted by *mvBag.containers.image.commands*.

```
[mvBag.containers.image]
forcePull = true
id = "ubuntu:16.04"
kind = "Docker"

  [[mvBag.containers.image.labels]]
  lastUpdateBy = "root"

  [[mvBag.containers.image.commands]]
  #install git
  Rochelle = ["install","y", "git core"]
   #install python
  run = ["install",python-pip"]

  [[mvBag.containers.image.commands]]
  #pull the python app
  runShell = ["git","clone", "https://git.logicpp.net/app/python-app.git"]
  #run the python app
  run = "python app/python-app.py"
```

Listing 5.4: mvBag.containers.image specification

mvBag.containers.networks is used to open specified ports on the node on which a container will be running.

```
[[mvBag.containers.networks]]
containerPort = 80
hostPort = 80
name = "frontEndPort"
protocol = "TCP"
```

Listing 5.5: mvBag.containers.networks specification

`velo` supports ephemeral volumes, which are defined at the container or pod level. The definition must include the *name*, *mount path* and whether its *read only*.

```
[[mvBag.containers.volumeMounts]]
mountPath = "/docker_storage/booking"
name = "bookingFolder"
readOnly = false
```

Listing 5.6: mvBag.containers.volumeMounts specification

The full specification of the application is shown in **??**.

5.2 Syntax Illustration

Consider a travel application which is composed of four microservices, *UI Service*, *Car Reservation Service*, *Hotel Reservation Service* and *Flight Reservation Service*. Figure 5.2 depicts how the services are interconnected. In sections 5.2.1 and 5.2.2, we present the abstract specification of both containerisation and orchestration of this application using `velo`. Here, we use the `TOML` (tom,) file format, a minimal configuration file format compatible with `YAML`.

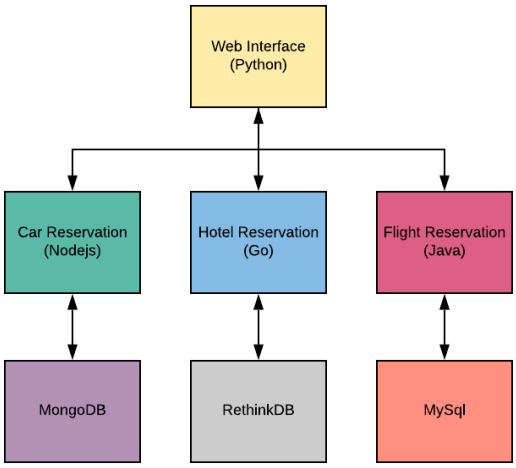

Figure 5.2: Travel App

5.2.1 `velo` – Rich Specification

In this scenario, we package each component in its own container. Thus, we use a *virtual bag (mvBag)* that contains six containers. Listing B.1 presents the rich specifications of the deployment for the travel application depicted in Figure 5.2.

mvBag here denotes the coherent unit introduced in section 2.3, known in various tools as a *Pod*. The *mvBag* has *id*, which uniquely identfies the application, the expected number of instances of the application is denoted by *instances*. If one needs other applications to communicate with the current one, the network names are defined using *networks*.

matchExpressions is a grouping mechanism which groups containers according to the specified expression. *user* denotes the account from which the application will be running. *mvBag.labels* assigns labels to the *mvBag* which can be used to as an identity mechanism once the application is deployed to apply other services, such restarting pods who match these labels.

One *mvBag* can contain multiple containers denoted by *mvBag.containers*. A single *mvBag.containers* maps to the coherent unit introduced in section 2.1. Health checks may be specified per container to be run against that container, denoted by *mvBag.containers.healthcheck*. As depicted in Figure 2.1 a container is an instance of an Image, here denoted by *mvBag.containers.image*. An Image is built from a set of instructions denoted by *mvBag.containers.image.commands*.

mvBag.containers.networks is used to open specified ports on the node on which a container will be running. `velo` supports ephemeral volumes, which are defined at the container or pod level. The definition must include the *name*, *mount path* and whether its *read only*.

```
author = "John Doe john@example.com"

[mvBag]
id = "travelwebapp"
instances = 2
networks = ["Net1","Net2"]
matchExpressions = [["app","In","travelwebapp"],["environment","In","production"]]
user = "gervasius"
version = "0.1"

  [[mvBag.labels]]
  App = "travelwebapp"
  Environment = "production"

  [[mvBag.containers]]
  name = "webinterface"
  restart = "Always"
  user = "admin"

    [mvBag.containers.healthcheck]
    gracePeriodSeconds = 30
    ignoreHttp1xx = false
    intervalSeconds = 30
    maxConsecutiveFailures = 3
    path = "somePath"
    portIndex = 1
    protocol = "TCP"
    timeoutSeconds = 60

    [mvBag.containers.image]
    forcePull = true
    id = "ubuntu:16.04"
    kind = "Docker"

      [[mvBag.containers.image.labels]]
      lastUpdateBy = "root"

      [[mvBag.containers.image.commands]]
      runShell = ["install","y","git core"]
      run = ["install","y","pythonpip"]


      [[mvBag.containers.image.commands]]
      runShell = ["git","clone","https://git.logicpp.net/app/pythonapp.git"]
      run = "python app/pythonapp.py"

    [mvBag.containers.resources]
    cpus = 4
    mem = 32

    [[mvBag.containers.labels]]
    tier = "frontend"
    App = "travelwebapp"

    [[mvBag.containers.volumeMounts]]
    mountPath = "/docker_storage/booking"
    name = "bookingFolder"
    readOnly = false

  [[mvBag.containers]]
  name = "carreservation"
  restart = "OnFailure"
  user = "root"
```

```
64
65      [mvBag.containers.image]
66      forcePull = true
67      id = "node:10"
68      kind = "Docker"
69
70        [[mvBag.containers.image.commands]]
71        workdir = "/usr/src/app"
72        run = "npm install"
73        copy = ["./package.json","/release"]
74        entrypoint = ["sh","run.sh"]
75
76      [[mvBag.containers.labels]]
77      tier = "backend"
78      App = "travelwebapp"
79
80      [[mvBag.containers.networks]]
81      containerPort = 8080
82      hostPort = 8080
83      name = "nodePort"
84      protocol = "TCP"
85
86    [[mvBag.containers]]
87    name = "hotelreservation"
88    restart = "OnFailure"
89    user = "root"
90
91      [mvBag.containers.image]
92      forcePull = true
93      id = "golang:1.12alpine3.9"
94      kind = "Docker"
95
96        [[mvBag.containers.image.commands]]
97        workdir = "/usr/src/app"
98        copy = [".","/app"]
99
100       [[mvBag.containers.image.commands]]
101       run = "go build o main ."
102
103     [[mvBag.containers.labels]]
104     tier = "backend"
105     App = "travelwebapp"
106
107     [[mvBag.containers.networks]]
108     containerPort = 8081
109     hostPort = 8081
110     name = "GoPort"
111     protocol = "TCP"
112
113   [[mvBag.containers]]
114   name = "flightreservation"
115   restart = "OnFailure"
116   user = "root"
117
118     [mvBag.containers.image]
119     forcePull = true
120     id = "openjdk:8jdkalpine"
121     kind = "Docker"
122
123       [[mvBag.containers.image.commands]]
124       workdir = "/usr/src/app"
125       copy = ["./application.jar","/app"]
126       entrypoint = ["/usr/bin/java"]
127
128       [[mvBag.containers.image.commands]]
129       runShell = ["jar","/usr/src/app/application.jar"]
130
131     [[mvBag.containers.labels]]
132     tier = "backend"
133     App = "travelwebapp"
134
135     [[mvBag.containers.networks]]
136     containerPort = 8888
137     hostPort = 8888
138     name = "javaPort"
139     protocol = "TCP"
140
141     [[mvBag.containers.volumeMounts]]
142     mountPath = "/var/lib/configrepo"
143     name = "configFolder"
144     readOnly = false
145
146   [[mvBag.containers]]
147   name = "mongoDB"
```

```
148  restart = "OnFailure"
149  user = "root"
150
151    [mvBag.containers.image]
152    forcePull = true
153    id = "ubuntu:10.4"
154    kind = "Docker"
155
156      [[mvBag.containers.image.commands]]
157      workdir = "/data"
158      run = "aptget install y mongodborg"
159
160      [[mvBag.containers.image.commands]]
161      runShell = ["mongod"]
162
163    [[mvBag.containers.labels]]
164    tier = "storage"
165    App = "travelwebapp"
166
167    [[mvBag.containers.networks]]
168    containerPort = 27017
169    hostPort = 27017
170    name = "mongoProcess"
171    protocol = "TCP"
172
173    [[mvBag.containers.networks]]
174    containerPort = 28017
175    hostPort = 28017
176    name = "mongoHttp"
177    protocol = "TCP"
178
179    [[mvBag.containers.volumeMounts]]
180    mountPath = "/data"
181    name = "dataFolder"
182    readOnly = false
183
184  [[mvBag.containers]]
185  name = "RethinkDB"
186  restart = "OnFailure"
187  user = "root"
188
189    [mvBag.containers.image]
190    forcePull = true
191    id = "ubuntu:10.4"
192    kind = "Docker"
193
194      [[mvBag.containers.image.commands]]
195      workdir = "/data"
196      run = "aptget install y rethinkdb pythonpip"
197
198      [[mvBag.containers.image.commands]]
199      runShell = ["pip","install","rethinkdb"]
200
201    [[mvBag.containers.labels]]
202    tier = "storage"
203    App = "travelwebapp"
204
205    [[mvBag.containers.networks]]
206    containerPort = 8080
207    hostPort = 8080
208    name = "WebUI"
209    protocol = "TCP"
210
211    [[mvBag.containers.networks]]
212    containerPort = 28015
213    hostPort = 28015
214    name = "rethink process"
215    protocol = "TCP"
216
217    [[mvBag.containers.networks]]
218    containerPort = 29015
219    hostPort = 29015
220    name = "rethink cluster"
221    protocol = "TCP"
222
223    [[mvBag.containers.volumeMounts]]
224    mountPath = "/data"
225    name = "dataFolder"
226    readOnly = false
227
228  [[mvBag.containers]]
229  name = "MySql"
230  restart = "OnFailure"
231  user = "root"
```

```
232
233     [mvBag.containers.image]
234     forcePull = true
235     id = "ubuntu:10.4"
236     kind = "Docker"
237
238       [[mvBag.containers.image.commands]]
239       workdir = "/data"
240       run = "aptget install y mysqlserver"
241
242       [[mvBag.containers.image.commands]]
243       runShell = ["mysqld_safe"]
244
245     [[mvBag.containers.labels]]
246     tier = "storage"
247     App = "travelwebapp"
248
249     [[mvBag.containers.networks]]
250     containerPort = 3306
251     hostPort = 3306
252     name = "mysqlPort"
253     protocol = "TCP"
254
255     [[mvBag.containers.volumeMounts]]
256     mountPath = "/etc/mysql"
257     name = "dataFolder"
258     readOnly = false
259
```

Listing 5.7: Travel App Specification – Rich

5.2.2 `velo` – Light Specification

The specifications presented in section 5.2.1 can also be presented in a leaner fashion, as shown in Listing 5.8. In a light specification, only the information that could not be guessed or configured in different possible ways is provided.

```
1  author = "John Doe john@example.com"
2  [mvBag]
3    id = "travelwebapp"
4    instances = 2
5    networks = ["Net1","Net2"]
6    matchExpressions = [["app","In","travelwebapp"],[ "environment","In","production"]]
7
8  user = "gervasius"
9  version = "0.1"
10
11   [[mvBag.labels]]
12   App = "travelwebapp"
13   Environment ="production"
14
15   [[mvBag.containers]]
16   name = "webinterface"
17
18     [mvBag.containers.image]
19     forcePull = true
20     id = "ubuntu:16.04"
21     kind = "Docker"
22
23       [[mvBag.containers.image.commands]]
24       runShell = ["install","y", "git core"]
25       run = ["install","y", "pythonpip"]
26
27       [[mvBag.containers.image.commands]]
28       runShell = ["git","clone", "https://git.logicpp.net/app/pythonapp.git"]
29       run = "python app/pythonapp.py"
30
31     [[mvBag.containers.labels]]
32     tier = "frontend"
33     App = "bookingwebapp"
34
35   [[mvBag.containers]]
36   name = "carreservation"
37
38     [mvBag.containers.image]
39     forcePull = true
40     id = "nginx:1.15.8alpine"
```

```
41      kind = "Docker"
42
43        [[mvBag.containers.image.commands]]
44        workdir = "/usr/src/app"
45        #copy package.json
46        run = "npm install"
47        copy = ["./package.json","/release"]
48        entrypoint = ["sh", "run.sh"]
49
50      [[mvBag.containers.labels]]
51      tier = "backend"
52      App = "travelwebapp"
53
54    [[mvBag.containers]]
55    name = "hotelreservation"
56
57      [mvBag.containers.image]
58      forcePull = true
59      id = "golang:1.12.0alpine3.9"
60      kind = "Docker"
61
62        [[mvBag.containers.image.commands]]
63        workdir = "/usr/src/app"
64        copy = [".","/app"]
65
66        [[mvBag.containers.image.commands]]
67        #we run go build to compile the binary
68        run = "go build o main ."
69
70      [[mvBag.containers.labels]]
71      tier = "backend"
72      App = "travelwebapp"
73
74    [[mvBag.containers]]
75    name = "flightreservation"
76
77      [mvBag.containers.image]
78      forcePull = true
79      id = "openjdk:8jdkalpine"
80      kind = "Docker"
81
82        [[mvBag.containers.image.commands]]
83        workdir = "/usr/src/app"
84        copy = ["./application.jar", "/app"]
85        entrypoint = ["/usr/bin/java",""]
86
87        [[mvBag.containers.image.commands]]
88        runShell = ["jar", "/usr/src/app/application.jar"]
89
90      [[mvBag.containers.labels]]
91      tier = "backend"
92      App = "travelwebapp"
93
94    [[mvBag.containers]]
95    name = "mongoDB"
96
97      [mvBag.containers.image]
98      forcePull = true
99      id = "ubuntu:10.4"
100     kind = "Docker"
101
102       [[mvBag.containers.image.commands]]
103       workdir = "/data"
104       run = "aptget install y mongodborg"
105
106       [[mvBag.containers.image.commands]]
107       runShell = ["mongod"]
108
109     [[mvBag.containers.labels]]
110     tier = "storage"
111     App = "travelwebapp"
112
113   [[mvBag.containers]]
114   name = "RethinkDB"
115
116     [mvBag.containers.image]
117     forcePull = true
118     id = "ubuntu:10.4"
119     kind = "Docker"
120
121       [[mvBag.containers.image.commands]]
122       workdir = "/data"
123       run = "aptget install y rethinkdb pythonpip"
124
```

```
125       [[mvBag.containers.image.commands]]
126       # Install python driver for rethinkdb
127       runShell = ["pip","install","rethinkdb"]
128
129     [[mvBag.containers.labels]]
130     tier = "storage"
131     App = "travelwebapp"
132
133   [[mvBag.containers]]
134   name = "MySql"
135
136     [mvBag.containers.image]
137     forcePull = true
138     id = "ubuntu:10.4"
139     kind = "Docker"
140
141       [[mvBag.containers.image.commands]]
142       workdir = "/data"
143       run = "aptget install y mysqlserver"
144
145       [[mvBag.containers.image.commands]]
146       runShell = ["mysqld_safe"]
147
148     [[mvBag.containers.labels]]
149     tier = "storage"
150     App = "travelwebapp"
151
```

Listing 5.8: Travel App Specification – Light

Once the desired state of a deployment is specified in `velo`, we pass it to the source-to-source compiler. During the compilation process, we indicate the specific container and orchestration tools that we target. In section 5.4, we discuss the compilation process and further illustrate it.

5.3 The Semantics

The purpose of this section is not to introduce the formal semantics of `velo` to reason about its execution. Instead, the semantics aims to provide a semi-formal hint about the meaning of the key concepts (virtual bag, container) introduced for abstraction in `velo`. From the meaning of these concepts, the behaviour of the underlying tools and technologies actually running these concepts is discussed.

5.3.1 Definitions

Let $\mathcal{H}_i$ denote a host; $\mathcal{H}_i$ could be a physical or a virtual machine. Let $\mathbb{C}$ denote a cluster of hosts $\mathcal{H}_0, \ldots, \mathcal{H}_n$. We note that $\mathbb{C} = \{\mathcal{H}_0, \ldots, \mathcal{H}_n\}$. We define a virtual bag v as a tuple consisting of two components: the configurations cn_{v} and a collection of containers running within the bag. We note $\mathsf{v} = \langle cn_{\mathsf{v}}, \{\mathrm{c}_0, \mathrm{c}_1, \ldots, \mathrm{c}_\ell\}\rangle$. As well, we define a container c_- as a tuple of three components: its configurations cn_{c_-}, the image the container is built upon im_{c_-} and finally the processes running within the container. We note $\mathrm{c}_i = \langle cn_{\mathrm{c}_i}, im_{\mathrm{c}_-}, \{\mathsf{p}_0, \mathsf{p}_1, \ldots, \mathsf{p}_k\}\rangle$. The reader should note that the configurations of both a virtual bag and a container encompass all the attributes used to describe these concepts. Note that the processes making up a container do not include the processes involved in the inner working of

the container. Rather, they include the processes corresponding to the tools and applications running under the container.

5.3.2 The Behavior

As indicated earlier in this document, our specification only describes the desired state of containerisation and orchestration of the production infrastructure. Furthermore, both containerisation and orchestration have specific tools and technologies devoted to them. As such, `velo` does not have to worry about the semantics of the underlying tools. Rather, we discuss the behaviour of the underlying tools as `velo` expresses the desired state.

We introduce the *execute* predicate to discuss the behaviour expected of a subset of a cluster as a virtual bag is being executed. Given a virtual bag $\mathtt{v}_j = \langle cn_{\mathtt{v}_j}, \{\mathrm{c}_0, \mathrm{c}_1, \ldots, \mathrm{c}_\mathrm{m}\}\rangle$, Equation 1 presents the behavior. While a virtual bag is being executed, first, the configuration of the virtual bag needs to be enforced. As well, for each container orchestrated by the virtual bag should always be running until it completes. The reader should note that here, we use Linear Temporal Logic (LTL) (Babenyshev and Rybakov, 2011). In Equation 1, the *orchestrated_by* predicate returns all containers being orchestrated by that virtual bag. As well the *runs* and *completed* predicates refer to activities of containers which we shall define shortly.

$$\begin{aligned} execute(\mathtt{v}_j) \models \Box((enforce(cn_{\mathtt{v}_j})) \wedge (\forall \mathrm{c}_\imath \\ \in orchestrated\_by(\mathtt{v}_j) \cdot runs(\mathrm{c}_\imath)\ U\ completed(\mathrm{c}_\imath))) \end{aligned} \tag{5.1}$$

Equation 1: Execution of a virtual bag

Equation 2 sheds light on the semantics of a running container. The latter should instantiate the image set in the specification of the container, enforce the configuration set for it and guarantee that sometime in the future each process corresponding to applications and services hosted by the container is running. As well, Equation 3 shows when a container completes. Once a container completes, it is always true that all processes forming part of the container are all done.

$$\begin{aligned} run(\mathrm{c}_\imath) \models (inst\_image(im_{\mathrm{c}_\imath})\ \wedge (\circ(enforce(cn_{\mathrm{c}_\imath})\ \wedge (\forall \mathtt{p}_\ell \\ \in hosted\_by(\mathrm{c}_\imath) \cdot \Diamond(exec\_proc(\mathtt{p}_\ell)))))) \end{aligned} \tag{5.2}$$

Equation 2: Running Container

$$completed(\mathrm{c}_\iota) \models \Box(\forall \mathsf{p}_\ell \in hosted\_by(\mathrm{c}_\iota) \cdot done(\mathsf{p}_\ell)) \tag{5.3}$$

Equation 3: Completed Container

5.4 Source-to-Source Compiler

5.4.1 Mechanism

The purpose of our compilation is not to generate *machine code*. Instead, it is to transform the **abstract** specification of the desired state of an application deployment both regarding containerisation and orchestration into a more **specific** description or specification for a given containerisation or orchestration technology, and thus the configuration language that technology supports. As depicted in Figure 5.3, `velo` currently supports `kubernetes`, `Docker Compose` and `Mesos Marathon` as target orchestration technology/-tool and `Docker` for containerisation. As such, our transpiler can generate files in `Dockerfile`, `yaml` and `json` formats.

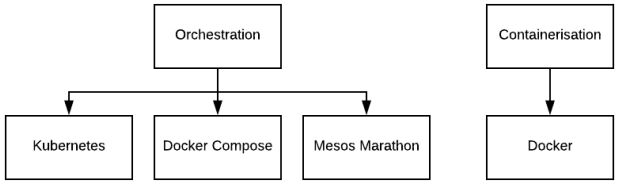

Figure 5.3: Containerisation and Orchestration tools

We defined `velo`'s grammar in `antlr` (Parr and Quong, 1995; Parr and Fisher, 2011), an *adaptive* LL parser (Sippu and Soisalon-Soininen, 1982), and implemented the rules in `GoLang`[2]. The reader should note that the rules we are referring to here are not related to the BNF production rules discussed in section 5.1. Instead, they refer to the actions expected by `antlr`.

`antlr` distinguishes three compilation phases: *lexical analysis* (lexer), *parsing* and *tree walking*. The transpiler first uses the lexer to analyse the input file (our `toml` specification). The lexer groups the characters from the input file into a stream of tokens. The parser then parses the stream and generates a tree of tokens, the parse tree. The latter should be understood as a concrete representation of the input according to the grammar.

Rather than using the parse tree, we generate an Abstract Syntax Tree (AST), a finite, directed, labelled tree, where each node represents a lan-

[2] `https://golang.org/`

guage construct and its children represent components of the construct. The AST provides more meaningful constructs rather than pure artefacts from the grammar. It is dense, without unnecessary nodes and subsequently easy to walk. `antlr` classifies these nodes into *terminal* and *non-terminal* nodes. A terminal node refers to a node without children, while a *non-terminal* node refers to a node with children. Figure 5.4 shows an example of AST for `velo`.

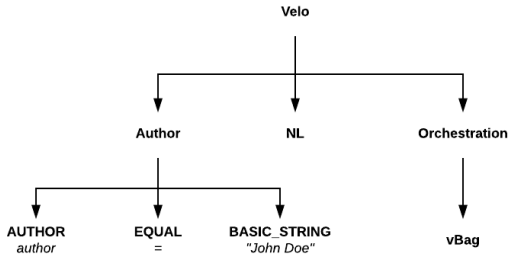

Figure 5.4: A Simple AST for Velo

Based on the generated AST, the transpiler will then generate the specification file for the selected container and orchestration tools as discussed earlier. The various compilation phases are depicted in the diagram in Figure 5.5.

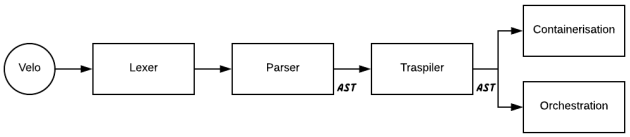

Figure 5.5: Compilation Process

The final step, the generation of the target specification file, requires the traversal of the AST. By default, `antlr` provides a *visitor* pattern (Gamma et al., 1995) for tree traversal. However, we opted to implement a *listener* pattern. The latter automates the call for each specific module whenever an event related to the listener occurs, while the visitor pattern requires to visit each node to ensure they do not get omitted during the generation process.

5.4.2 Error Handling

By default, `antlr` checks the input against the grammar syntax. If there are any inconsistencies, the corresponding error information is printed in the console, and the string is returned as recognised by `antlr`. In reality,

`antlr` replaces the unrecognised nodes with 'undefined'. In our compiler, however, we introduce a more customised error handling approach by overriding `antlr`'s default error listener. We throw a syntax error exception for any syntax error encountered and saving exception information to a log file and printing to the console detailed useful information, including the line and column number in the input file causing the error. This assists the user in addressing the syntax error correctly and restart the process. In order to save users' time during the debugging process, all errors are reported to the user during each compilation round.

5.4.3 Illustration

To illustrate the transpiler in `velo`, consider the abstract specification discussed in section 5.2.1. The corresponding `toml` file is passed to the transpiler in `velo` and the target tools are specified as depicted in Figure 5.6. Where *-f* denotes the path to the file containing `velo` specifications, *-doc* denotes whether to output a Dockerfile, *-kub* denotes whether to output Kubernetes configuration files,*-com* denotes whether to output Docker Compose configuration files and *-mar* denotes whether to output Marathon configuration files.

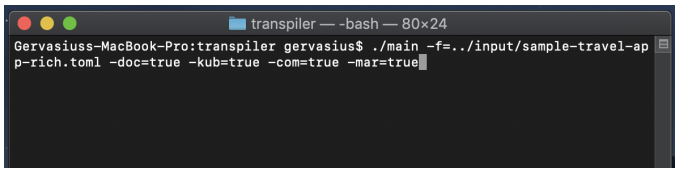

Figure 5.6: Transpiler Options

If the light specification is passed to the transpiler, information that could not be guessed or configured in different possible ways is then interactively prompted from the user as depicted in Figure 5.7.

Below we present the generated `Dockerfiles` followed by generated orchestration files for `kubernetes`, `Docker Compose` and `Mesos` presented in Listings 5.16, 5.17 and 5.18. As this illustration demonstrates, from one single abstract specification of the desired state of the deployment, one can now automatically produce both orchestration and containerisation specifications for various tools.

```
FROM ubuntu:16.04
LABEL maintainer = "John Doe john@example.com"
LABEL description = "web-interface"
LABEL version = "0.1"
RUN [install ,-y,git core ]
RUN ["install", "-y", "python-pip"]
RUN  [git ,clone ,https://git.logicpp.net/app/python-app.git ]
RUN python app/python-app.py
VOLUME bookingFolder /docker_storage/booking
```

Listing 5.9: Dockerfile for the Web UI Service

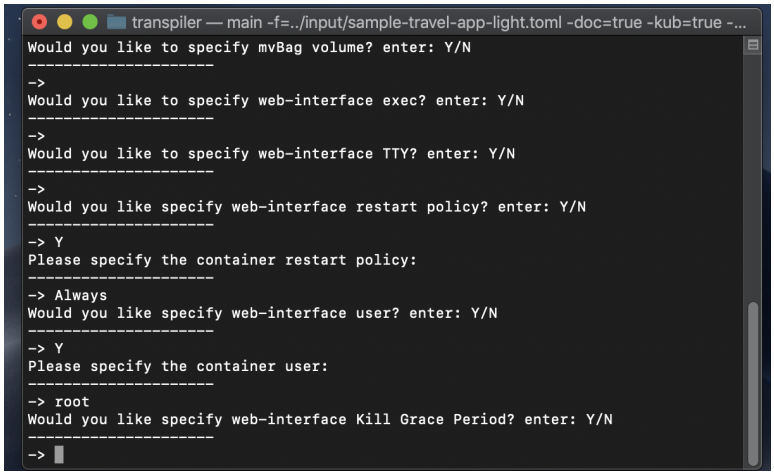

Figure 5.7: Velo light interaction

```
FROM node:10
LABEL maintainer = "John Doe john@example.com"
LABEL description = "car-reservation"
LABEL version = "0.1"
EXPOSE 8080
WORKDIR /usr/src/app
RUN npm install
COPY ./package.json /release
ENTRYPOINT [sh ,run.sh]
```

Listing 5.10: Dockerfile for Car Reservation Service

```
FROM golang:1.12.0-alpine3.9
LABEL maintainer = "John Doe john@example.com"
LABEL description = "hotel-reservation"
LABEL version = "0.1"
EXPOSE 8081
WORKDIR /usr/src/app
COPY . /app
RUN go build -o main .
```

Listing 5.11: Dockerfile for Hotel Reservation Service

```
FROM openjdk:8-jdk-alpine
LABEL maintainer = "John Doe john@example.com"
LABEL description = "flight-reservation"
LABEL version = "0.1"
EXPOSE 8888
WORKDIR /usr/src/app
COPY ./application.jar /app
ENTRYPOINT [/usr/bin/java , ,]
RUN  [-jar ,/usr/src/app/application.jar ]
VOLUME configFolder /var/lib/config-repo
```

Listing 5.12: Dockerfile for Flight Reservation Service

```
FROM ubuntu:10.4
LABEL maintainer = "John Doe john@example.com"
LABEL description = "mongoDB"
LABEL version = "0.1"
EXPOSE 27017
EXPOSE 28017
WORKDIR /data
RUN apt-get install -y mongodb-org
RUN  [mongod]
VOLUME dataFolder /data
```

Listing 5.13: Dockerfile for MongoDB

```
FROM ubuntu:10.4
LABEL maintainer = "John Doe john@example.com"
LABEL description = "RethinkDB"
LABEL version = "0.1"
EXPOSE 8080
EXPOSE 28015
EXPOSE 29015
WORKDIR /data
RUN apt-get install -y rethinkdb python-pip
RUN  [pip ,install ,rethinkdb ]
VOLUME dataFolder /data
```

Listing 5.14: Dockerfile for RethinkDB

```
FROM ubuntu:10.4
LABEL maintainer = "John Doe john@example.com"
LABEL description = "MySql"
LABEL version = "0.1"
EXPOSE 3306
WORKDIR /data
RUN apt-get install -y mysql-server
RUN  [mysqld_safe]
VOLUME dataFolder /etc/mysql
```

Listing 5.15: Dockerfile for MySql

```
apiVersion: v1
kind: Deployment
metadata:
  name: travel web app
  labels:
  - App: travel web app
  - Environment: production
spec:
  replicas: "2"
  selector:
    matchExpressions: [{key: app, operator: In, values: [ travel web app ]}, {key:
     environment ,
        operator: In, values: [ production ]}]
  template:
    metadata:
      name: travel web app
      labels:
      - App: travel web app
      - Environment: production
    containers:
    - name: web interface
      image: ubuntu:16.04
      restartPolicy: Always
      volumeMounts:
      - name: bookingFolder
        mountPath: /docker_storage/booking
      resources:
        limits:
        - cpu: "4"
          memory: "32"
    - name: car reservation
      image: node:10
      restartPolicy: OnFailure
      ports:
      - containerPort: "8080"
    - name: hotel reservation
```

```
        image: golang: 1.12.0  alpine3 .9
        restartPolicy: OnFailure
        ports:
        - containerPort: "8081"
      - name: flight  reservation
        image: openjdk: 8 jdk  alpine
        restartPolicy: OnFailure
        ports:
        - containerPort: "8888"
        volumeMounts:
        - name: configFolder
          mountPath: /var/lib/ config  repo
      - name: mongoDB
        image: ubuntu:10.4
        restartPolicy: OnFailure
        ports:
        - containerPort: "27017"
        - containerPort: "28017"
        volumeMounts:
        - name: dataFolder
          mountPath: /data
      - name: RethinkDB
        image: ubuntu:10.4
        restartPolicy: OnFailure
        ports:
        - containerPort: "8080"
        - containerPort: "28015"
        - containerPort: "29015"
        volumeMounts:
        - name: dataFolder
          mountPath: /data
      - name: MySql
        image: ubuntu:10.4
        restartPolicy: OnFailure
        ports:
        - containerPort: "3306"
        volumeMounts:
        - name: dataFolder
          mountPath: /etc/mysql
```

Listing 5.16: Orchestration for Kubernetes

```
version: "3"
services:
  service:
    MySql:
      image: ubuntu:10.4
      deploy:
        replicas: 2
        labels:
        - App: travel  web app
        - Environment: production
      containerName: MySql
      volumes:
      - /etc/mysql
      ports:
      - "3306"
      restartPolicy: OnFailure
      networks:
      - mysqlPort
      labels:
      - tier: storage
```

```
      - App: travel web app
  RethinkDB:
    image: ubuntu:10.4
    deploy:
      replicas: 2
      labels:
      - App: travel web app
      - Environment: production
    containerName: RethinkDB
    volumes:
    - /data
    ports:
    - "8080"
    - "28015"
    - "29015"
    restartPolicy: OnFailure
    networks:
    - WebUI
    - rethink process
    - rethink cluster
    labels:
    - tier: storage
    - App: travel web app
  car reservation:
    image: node:10
    deploy:
      replicas: 2
      labels:
      - App: travel web app
      - Environment: production
    containerName: car reservation
    ports:
    - "8080"
    restartPolicy: OnFailure
    networks:
    - nodePort
    labels:
    - tier: back end
    - App: travel web app
  flight reservation:
    image: openjdk: 8 jdk alpine
    deploy:
      replicas: 2
      labels:
      - App: travel web app
      - Environment: production
    containerName: flight reservation
    volumes:
    - /var/lib/config repo
    ports:
    - "8888"
    restartPolicy: OnFailure
    networks:
    - javaPort
    labels:
    - tier: back end
    - App: travel web app
  hotel reservation:
    image: golang: 1.12.0 alpine3 .9
    deploy:
      replicas: 2
      labels:
```

```
        - App: travel web app
        - Environment: production
      containerName: hotel reservation
      ports:
      - "8081"
      restartPolicy: OnFailure
      networks:
      - GoPort
      labels:
      - tier: back end
      - App: travel web app
    mongoDB:
      image: ubuntu:10.4
      deploy:
        replicas: 2
        labels:
        - App: travel web app
        - Environment: production
      containerName: mongoDB
      volumes:
      - /data
      ports:
      - "27017"
      - "28017"
      restartPolicy: OnFailure
      networks:
      - mongoProcess
      - mongoHttp
      labels:
      - tier: storage
      - App: travel web app
    web interface:
      image: ubuntu:16.04
      deploy:
        replicas: 2
        resources:
          limits:
          - cpus: "4"
        labels:
        - App: travel web app
        - Environment: production
      containerName: web interface
      volumes:
      - /docker_storage/booking
      restartPolicy: Always
      healthCheck:
        interval: "30"
        timeout: "60"
        retries: "3"
        start_period: "30"
      labels:
      - tier: front end
      - App: booking web app
```

Listing 5.17: Orchestration for Docker Compose

```
{
    "version": "2019-08-26T23:53:18+02:00",
    "id": "travel-web-app",
    "user": "root",
    "instances": 2,
    "networks": [
```

```
        "Net1",
        "Net2"
    ],
    "executorResources": {},
    "constraints": [
        [
            "app",
            "In",
            "travel-web-app"
        ],
        [
            "environment",
            "In",
            "production"
        ]
    ],
    "labels": [
        {
            "App": "travel-web-app"
        },
        {
            "Environment": "production"
        }
    ],
    "scaling": {
        "kind": "Docker"
    },
    "scheduling": {
        "backoff": {}
    },
    "containers": [
        {
            "name": "web-interface",
            "check": {
                "http": {},
                "tcp": {}
            },
            "healthcheck": {
                "gracePeriodSeconds": 30,
                "intervalSeconds": 30,
                "maxConsecutiveFailures": 3,
                "path": "somePath",
                "protocol": "TCP",
                "timeoutSeconds": 60
            },
            "image": {
                "forcePull": true,
                "id": "ubuntu:16.04",
                "kind": "Docker",
                "labels": [
                    {
                        "lastUpdateBy": "root"
                    }
                ]
            },
            "labels": [
                {
                    "tier": "front-end"
                },
                {
                    "App": "booking-web-app"
                }
```

```
            ],
            "lifecycle": {},
            "resources": {
                "cpus": 4,
                "mem": 32
            },
            "volumeMounts": [
                {
                    "mountPath": "/docker_storage/booking",
                    "name": "bookingFolder"
                }
            ]
        },
        {
            "name": "car-reservation",
            "check": {
                "http": {},
                "tcp": {}
            },
            "healthcheck": {},
            "image": {
                "forcePull": true,
                "id": "node:10",
                "kind": "Docker"
            },
            "labels": [
                {
                    "tier": "back-end"
                },
                {
                    "App": "travel-web-app"
                }
            ],
            "lifecycle": {},
            "resources": {},
            "networks": [
                {
                    "containerPort": 8080,
                    "hostPort": 8080,
                    "name": "nodePort",
                    "protocol": "TCP"
                }
            ]
        },
        {
            "name": "hotel-reservation",
            "check": {
                "http": {},
                "tcp": {}
            },
            "healthcheck": {},
            "image": {
                "forcePull": true,
                "id": "golang:1.12.0-alpine3.9",
                "kind": "Docker"
            },
            "labels": [
                {
                    "tier": "back-end"
                },
                {
                    "App": "travel-web-app"
```

```
                }
            ],
            "lifecycle": {},
            "resources": {},
            "networks": [
                {
                    "containerPort": 8081,
                    "hostPort": 8081,
                    "name": "GoPort",
                    "protocol": "TCP"
                }
            ]
        },
        {
            "name": "flight-reservation",
            "check": {
                "http": {},
                "tcp": {}
            },
            "healthcheck": {},
            "image": {
                "forcePull": true,
                "id": "openjdk:8-jdk-alpine",
                "kind": "Docker"
            },
            "labels": [
                {
                    "tier": "back-end"
                },
                {
                    "App": "travel-web-app"
                }
            ],
            "lifecycle": {},
            "resources": {},
            "networks": [
                {
                    "containerPort": 8888,
                    "hostPort": 8888,
                    "name": "javaPort",
                    "protocol": "TCP"
                }
            ],
            "volumeMounts": [
                {
                    "mountPath": "/var/lib/config-repo",
                    "name": "configFolder"
                }
            ]
        },
        {
            "name": "mongoDB",
            "check": {
                "http": {},
                "tcp": {}
            },
            "healthcheck": {},
            "image": {
                "forcePull": true,
                "id": "ubuntu:10.4",
                "kind": "Docker"
            },
```

```
            "labels": [
                {
                    "tier": "storage"
                },
                {
                    "App": "travel-web-app"
                }
            ],
            "lifecycle": {},
            "resources": {},
            "networks": [
                {
                    "containerPort": 27017,
                    "hostPort": 27017,
                    "name": "mongoProcess",
                    "protocol": "TCP"
                },
                {
                    "containerPort": 28017,
                    "hostPort": 28017,
                    "name": "mongoHttp",
                    "protocol": "TCP"
                }
            ],
            "volumeMounts": [
                {
                    "mountPath": "/data",
                    "name": "dataFolder"
                }
            ]
        },
        {
            "name": "RethinkDB",
            "check": {
                "http": {},
                "tcp": {}
            },
            "healthcheck": {},
            "image": {
                "forcePull": true,
                "id": "ubuntu:10.4",
                "kind": "Docker"
            },
            "labels": [
                {
                    "tier": "storage"
                },
                {
                    "App": "travel-web-app"
                }
            ],
            "lifecycle": {},
            "resources": {},
            "networks": [
                {
                    "containerPort": 8080,
                    "hostPort": 8080,
                    "name": "WebUI",
                    "protocol": "TCP"
                },
                {
                    "containerPort": 28015,
```

```
                    "hostPort": 28015,
                    "name": "rethink process",
                    "protocol": "TCP"
                },
                {
                    "containerPort": 29015,
                    "hostPort": 29015,
                    "name": "rethink cluster",
                    "protocol": "TCP"
                }
            ],
            "volumeMounts": [
                {
                    "mountPath": "/data",
                    "name": "dataFolder"
                }
            ]
        },
        {
            "name": "MySql",
            "check": {
                "http": {},
                "tcp": {}
            },
            "healthcheck": {},
            "image": {
                "forcePull": true,
                "id": "ubuntu:10.4",
                "kind": "Docker"
            },
            "labels": [
                {
                    "tier": "storage"
                },
                {
                    "App": "travel-web-app"
                }
            ],
            "lifecycle": {},
            "resources": {},
            "networks": [
                {
                    "containerPort": 3306,
                    "hostPort": 3306,
                    "name": "mysqlPort",
                    "protocol": "TCP"
                }
            ],
            "volumeMounts": [
                {
                    "mountPath": "/etc/mysql",
                    "name": "dataFolder"
                }
            ]
        }
    ]
}
```

Listing 5.18: Orchestration for Marathon Mesos

As this illustration demonstrates, from one single abstract specification of the desired state of the deployment, one can now automatically produce

both orchestration and containerisation specifications for various tools.

Chapter 6

Evaluation

To evaluate `velo`'s compiler correctness we grouped our tests in three categories:

- *Syntax testing* to verify that the lexer and the parser distinguish correct and incorrect rule sets;
- *Compiler testing* to verify that the shape of the resulting AST is the one we expect and;
- *Validation* to verify that the files generated by the compiler produce the correct artefacts when deployed to the corresponding tools.

6.1 Syntax testing

The basic idea is to create several input/output pairs for rules in a grammar and verify the expected output/result. The input can be a single line or multiple lines of strings or even an external file. The output can be only success or failure, an AST, or some text output which could be a rule's template return value.

We used Go testing framework [1] to write and execute our test cases. We chose to teach the parser to throw exceptions on errors which is done with custom error handling strategy. During compilation, our custom error handling strategy listens and collects all the syntax errors into a list. Therefore, we can verify the size of the error list against the expected result as depicted in listing 6.1.

```
1 package main
2
3 import (
4 "testing"
5 "../parser"
6 "./utils"
7 "github.com/antlr/antlr4/runtime/Go/antlr"
8 )
```

[1] `https://golang.org/pkg/testing/`

```
9
10 func TestGrammar(t *testing.T) {
11     tables := []struct {
12         filePath string
13         isValid  bool
14         }{
15             {"../input/samplesvbagrich.toml", true},
16             {"../input/samplemvbagrich.toml", true},
17             {"../input/samplesimpleerror.toml", false},
18             {"../input/sampleempty.toml", false},
19             }
20     for _, table := range tables {
21         errorListiner := utils.NewVeloErrorListener()
22         input, _ := antlr.NewFileStream(table.filePath)
23         lexer := parser.NewVeloLexer(input)
24         stream := antlr.NewCommonTokenStream(lexer, 0)
25         p := parser.NewVeloParser(stream)
26         p.RemoveErrorListeners()
27         p.AddErrorListener(errorListiner)
28         tree := p.Velo()
29         antlr.ParseTreeWalkerDefault.Walk(NewVeloListener(), tree)
30         if len(errorListiner.Errors) > 0 && table.isValid {
31             t.Errorf("Grammar was incorrect for %s , got: %t, expected: %t.", table.filePath, table.isValid,
      !table.isValid)
32         }
33         if len(errorListiner.Errors) == 0 && !table.isValid {
34             t.Errorf("Grammar was incorrect for %s , got: %t, expected: %t.", table.filePath, !table.isValid,
       table.isValid)
35             }
36     }
37 }
```

Listing 6.1: Grammar test

Test Case	Code Coverage	Test Result	Time
sample-svbag-rich.toml	29.3%	**PASS**	0.024s
sample-mvbag-rich.toml	29.8%	**PASS**	0.028s
sample-simple-error.toml	20.4%	**PASS**	0.028s
sample-empty.toml	1.1%	**PASS**	0.008s

Table 6.1: Results for syntax testing

6.2 Compiler testing

To verify that the shape of the resulting AST is the one we expect, we generate a JSON representation of the AST and verify that it is the same as the one expected. Each *terminal* node is converted into a JSON array that can contain other JSON objects. Each *non-terminal* node is converted into a JSON object that has two properties: *type*, which contains the integer value mapping to the type of token; while *text* contains the actual value of the token specified in the input file.

Therefore, given the AST depicted in Figure 6.1, we can deduce the JSON we expect as follows:

```
1     {
2         "copyCommand": [
3           {
4             "type": 215,
5             "text": "copy"
6           },
```

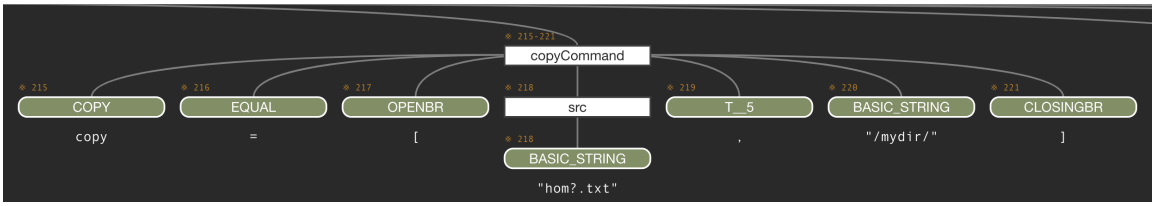

Figure 6.1: Copy command AST

```
7           {
8             "type": 216,
9             "text": "\u003d"
10          },
11          {
12            "type": 217,
13            "text": "["
14          },
15          {
16            "src": [
17              {
18                "type": 218,
19                "text": "\"hom?.txt\""
20              }
21            ]
22          },
23          {
24            "type": 219,
25            "text": ","
26          },
27          {
28            "type": 220,
29            "text": "\"/mydir/\""
30          },
31          {
32            "type": 221,
33            "text": "]"
34          }
35        ]
36      }
```

Listing 6.2: AST representation in JSON

We used JUnit 5 [2] testing framework to write and execute our test cases. We compare the expected JSON to the JSON generated from compilation using the built-in assertion mechanism, as shown in Listing 6.3.

```
1   package maintest;
2
3   import org.antlr.v4.runtime.ANTLRInputStream;
4   import org.antlr.v4.runtime.CommonTokenStream;
5   import org.antlr.v4.runtime.Lexer;
6   import org.apache.commons.io.IOUtils;
7   import org.junit.jupiter.api.Test;
8   import parser.VeloLexer;
9   import parser.VeloParser;
10  import utils.Utils;
11  import static org.junit.jupiter.api.Assertions.*;
```

[2] https://junit.org/junit5/

```
12  import java.io.IOException;
13  import java.io.InputStream;
14  import java.nio.charset.Charset;
15
16  public class CompilerTest {
17
18      @Test
19      void compilerTest() throws IOException {
20          Lexer lexer = new VeloLexer(new ANTLRInputStream(getClass().getResourceAsStream("/samplesvbagrich.
      toml")));
21          CommonTokenStream tokens = new CommonTokenStream(lexer);
22          VeloParser parser = new VeloParser(tokens);
23          String input = Utils.toJson(parser.velo(),true);
24          String expected = convertInputStreamToString(getClass().getResourceAsStream("/samplesvbagrich.json"
      ));
25          assertEquals(input,expected);
26      }
27
28      public String convertInputStreamToString(InputStream inputStream) throws IOException {
29          return IOUtils.toString(inputStream, Charset.defaultCharset());
30      }
31  }
```

Listing 6.3: Compiler test

Test Case	Code Coverage	Test Result	Time
sample-svbag-rich.toml	29.3%	**PASS**	0.193s
sample-mvbag-rich.toml	29.8%	**PASS**	0.2s
sample-simple-error.toml	20.4%	**FAIL**	0.224s

Table 6.2: Results for compiler testing

6.3 Validation

We deployed three virtual machines, each machine running on CentOS Linux (release 7.5.1884(Core)) OS. On $VM1$, we installed Docker CE 17.12.1-ce and Docker Compose 1.20.0. On $VM2$ we installed Minikube v1.3.1; a tool that makes it easy to run Kubernetes locally. $VM3$ we installed Marathon. The first step was to create Docker images from the Dockerfiles generated in listing 5.11, listing 5.10 and listing 5.9. Next, we ran Docker containers from the resulting images. Figure 6.2 shows the resulting images, and Figure 6.3 shows the resulting containers. Finally, we used Docker Compose to deploy the *YAML* file generated in listing 5.17. Figure 6.4 shows the resulting services.

To validate Kubernetes artefacts, we deployed the *YAML* generated in listing 5.16 on VM2. Figure 6.5 shows the resulting *Pod* and its *Containers*.

6.4 Discussion of results

The values in Tables 6.1 and 6.2 indicate the execution time, code coverage and test results of the executed tests described in Sections 6.1 and 6.2, respectively. As shown in Tables 6.1 and 6.2, the pass rate for the test

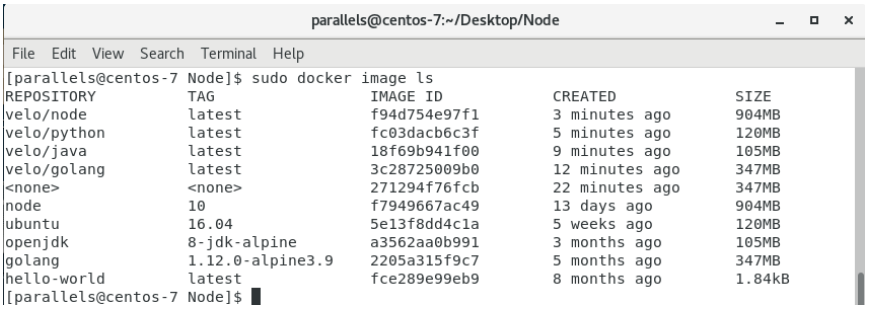

Figure 6.2: List of Docker images

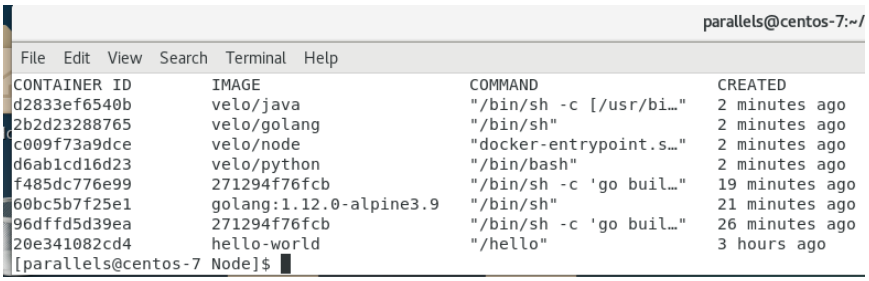

Figure 6.3: List of Docker containers

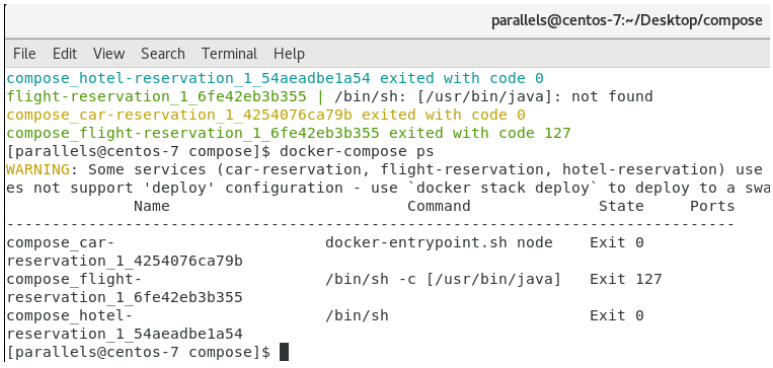

Figure 6.4: List of Docker compose services

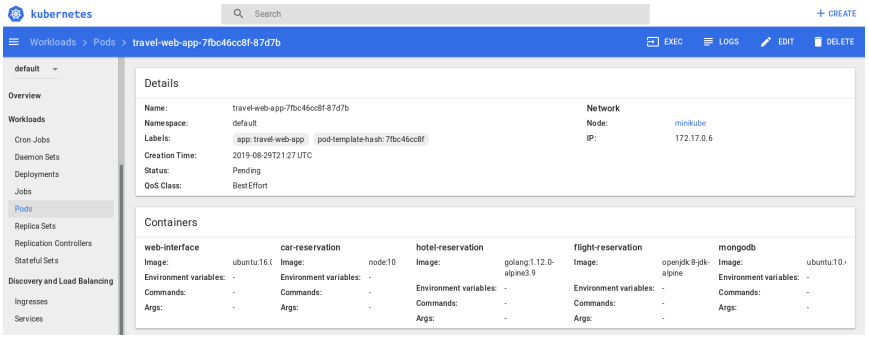

Figure 6.5: Minikube Dashboard

is 100% and 67%. These results can be interpreted as, `velo` **can distinguish between correct and incorrect syntax**. Furthermore, it can be interepreted, that, the generated AST should comply with the grammar for further processing by `velo`.

The values in Table 6.1 shows a low, average, code coverage of 26%. Code coverage is employed as a method to measure how thoroughly software is tested. Coverage is used by developers and vendors to indicate their confidence in the readiness of their software. Evaluating software coverage and taking proper actions lead to an improvement in software quality. It helps in evaluating the effectiveness of testing by providing data on different coverage items.

However, the reader should know that coverage techniques measure only one dimension of a multi-dimensional concept. Checks may exist for unexpected error conditions. Layers of code might obscure whether errors in low-level code propagate up to higher-level code. A developer might decide that handling all errors creates a more robust solution than tracing the possible errors.

Beyond addressing technical problems of the DSL and compiler, the results from Section 6.3 demonstrate that the configuration files output by the compiler, when deployed to the corresponding tools results into the artefacts understood by the said tools. Although, not fully integrated, developers running heterogeneous environments can utilise `velo` as unifying language to describe containerisation and orchestration requirements of complex applications.

Chapter 7

Conclusion

Despite efforts in standardisation and consolidation to streamline the ecosystem around containerisation and orchestration, there are still many challenges. Although the issue of application deployment in the cloud is now well understood, the existing solutions are faced with a challenging diversity and an inherent complexity.

In this research, we addressed the growing diversity in application containerisation and orchestration. These are two inter-related deployment and execution concepts which embrace different tools with different configuration environments. As a solution, we introduced `velo`, a unifying DSL that abstracts containerisation and orchestration in complex applications. `velo` has two components: (1) a language that helps specify an abstract desired state of the deployment in terms of containerisation and orchestration, and (2) a source-to-source compiler that automatically generates a specific target container and orchestration specifications. The `velo` DSL offers two specification modes. In the *rich* mode, the user can specify every single detail of the desired state of the orchestration and containerisation. In the *light* mode, all of the unnecessary details are removed, allowing the user to provide them during the compilation. In this document, we discussed both the DSL and the compiler and illustrated them using real-world examples.

The benefit of `velo` is that in one single abstract specification, both containerisation and orchestration concepts are defined and then automatically generated for specific technologies and tools. This allows a great level of flexibility between developer teams while maintaining a consistent core. It embraces the growing diversity around cloud application deployment, particularly regarding orchestration and containerisation. Finally, it enables us to enforce the best practices for each of these technologies.

In the future, we wish to extend the palette of target tools supported at the moment, particularly for containerisation. Moreover, although the compiler can distinguish between rich and light specifications, we wish to make the language even leaner and the compiler smarter. For example, currently, whenever the RUN command in a container refers to a tool, it goes

without saying that the latter should be automatically installed first and foremost. By the same token, whenever tools need to be installed an upgrade of the underlying distribution might be required before the installation. All these optimization rules will then be incorporated in the transpiler. Finally, to make adoption of the tool much easier, we wish to deploy it as a cloud service and have the users submit their abstract specification file and then retrieve and download the target files.

Bibliography

Cloudify. https://cloudify.co/. Accessed on May 31 2019.

Docker. https://www.docker.com/. Accessed on 02 Janaury 2018.

FreeBSD Jails. https://www.freebsd.org/cgi/man.cgi?query=jail&format=html. Accessed on 06 March 2018.

jclouds-docker. https://github.com/jclouds/jclouds/tree/master/apis/docker. Accessed on May 31 2019.

Kurbernetes. https://kubernetes.io/. Accessed on 26 February 2018.

Linux Containers - Introduction. https://linuxcontainers.org/lxc/introduction/. Accessed on 06 March 2018.

OpenVz. https://openvz.org/. Accessed on 06 March 2018.

Rkt. https://github.com/coreos/rkt. Accessed on 19 February 2018.

runc. https://github.com/opencontainers/runc/. Accessed on 06 March 2018.

Solaris Zones. https://docs.oracle.com/cd/E26502_01/html/E29024/preface-1.html. Accessed on 06 March 2018.

TOCSA. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca. Accessed on 8 June 2019.

TOML. https://github.com/toml-lang/toml. Accessed on 8 June 2019.

Babenyshev, S. and Rybakov, V. (2011). Linear temporal logic ltl: Basis for admissible rules. *Journal of Logic and Computation*, 21(2):157–177.

Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., and Wagner, S. (2013). OpenTOSCA – a runtime for TOSCA-based cloud applications. In *11^{th} International Conference on Service-Oriented Computing*, LNCS. Springer.

Binz, T., Breitenbücher, U., Kopp, O., and Leymann, F. (2014). chapter TOSCA: Portable Automated Deployment and Management of Cloud Applications, pages 527–549. Springer, New York.

Briggs, I., Day, M., Guo, Y., Marheine, P., and Eide, E. (2014). A performance evaluation of unikernels. In *Technical Report*.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, omega, and kubernetes. *ACM Queue*, 14:70–93.

Buyya, R., Rodriguez, M. A., Toosi, A. N., and Park, J. (2018). Cost-efficient orchestration of containers in clouds: A vision, architectural elements, and future directions. In *Journal of Physics: Conference Series*, volume 1108, page 012001. IOP Publishing.

Cito, J., Leitner, P., Gall, H. C., Dadashi, A., Keller, A., and Roth, A. (2015). Runtime metric meets developer: building better cloud applications using feedback. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 14–27. ACM.

de Alfonso, C., Calatrava, A., and Moltó, G. (2017). Container-based virtual elastic clusters. *Journal of Systems and Software*, 127:1–11.

Di Martino, B., Cretella, G., and Esposito, A. (2015). Advances in applications portability and services interoperability among multiple clouds. *IEEE Cloud Computing*, 2(2):22–28.

Dragoni, N., Dustdar, S., Larsen, S. T., and Mazzara, M. (2017). Microservices: Migration of a mission critical system. *CoRR*, abs/1704.04173.

Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2016). Microservices: yesterday, today, and tomorrow. *CoRR*, abs/1606.04036.

Dua, R., Raja, A. R., and Kakadia, D. (2014). Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 610–614. IEEE.

Fazio, M., Celesti, A., Ranjan, R., Liu, C., Chen, L., and Villari, M. (2016). Open issues in scheduling microservices in the cloud. *IEEE Cloud Computing*, 3(5):81–88.

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*, pages 171–172.

Ferry, N., Chauvel, F., Rossini, A., Morin, B., and Solberg, A. (2013). Managing multi-cloud systems with cloudmf. In *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, pages 38–45. ACM.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Goncalves, G., Endo, P., Santos, M., Sadok, D., Kelner, J., Melander, B., and Mangs, J.-E. (2011). Cloudml: An integrated language for resource, service and request description for d-clouds. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 399–406. IEEE.

Hassan, H. (2017). Organisational factors affecting cloud computing adoption in small and medium enterprises (smes) in service sector. *Procedia Computer Science*, 121:976 – 981. CENTERIS 2017 - International Conference on ENTERprise Information Systems / ProjMAN 2017 - International Conference on Project MANagement / HCist 2017 - International Conference on Health and Social Care Information Systems and Technologies, CENTERIS/ProjMAN/HCist 2017.

Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA.

Junqueira, F. P., Reed, B. C., and Serafini, M. (2011). Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE.

Kratzke, N. (2017a). About microservices, containers and their underestimated impact on network performance. *CoRR*, abs/1710.04049.

Kratzke, N. (2017b). Smuggling multi-cloud support into cloud-native applications using elastic container platforms. In *CLOSER*, pages 29–42.

Leitner, P., Cito, J., and Stöckli, E. (2016). Modelling and managing deployment costs of microservice-based cloud applications. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 165–174. ACM.

Linthicum, D. S. (2016). Moving to autonomous and self-migrating containers for cloud applications. *IEEE Cloud Computing*, 3(6):6–9.

Madhavapeddy, A. and Scott, D. J. (2013). Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30–44.

Maskasky, M. (2014). On standardization of containers.

McCracken, D. D. and Reilly, E. D. Backus-naur form (bnf). In *Encyclopedia of Computer Science*, pages 129–131. John Wiley and Sons Ltd., Chichester, UK.

Molnár, E., Molnár, R., Kryvinska, N., and Greguš, M. (2014). Web intelligence in practice. *Journal of Service Science Research*, 6(1):149–172.

Nardelli, M., Hochreiner, C., and Schulte, S. (2017). Elastic provisioning of virtual machines for container deployment. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 5–10. ACM.

Netto, H. V., Lung, L. C., Correia, M., Luiz, A. F., and de Souza, L. M. S. (2017). State machine replication in containers managed by kubernetes. *Journal of Systems Architecture*, 73:53–59.

Oliveira, T., Thomas, M., and Espadanal, M. (2014). Assessing the determinants of cloud computing adoption: An analysis of the manufacturing and services sectors. *Information & Management*, 51(5):497 – 510.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319.

Palma, D., Rutkowski, M., and Spatzier, T. (2016). Tosca simple profile in yaml version 1.0. *OASIS Committee Specification*, 1.

Parr, T. J. and Quong, R. W. (1995). Antlr: A predicated-ll(k) parser generator. *Softw. Pract. Exper.*, 25(7):789–810.

Parr, T. P. and Fisher, K. (2011). Ll(*): the foundation of the ANTLR parser generator. In *PLDI*, pages 425–436. ACM.

Pavlicek, R. (2016). *Unikernels*. O'Reilly Media, Incorporated.

Peinl, R. and Holzschuher, F. (2015). The docker ecosystem needs consolidation. In *CLOSER*, pages 535–542.

Peinl, R., Holzschuher, F., and Pfitzer, F. (2016). Docker cluster management for the cloud-survey results and own solution. *Journal of Grid Computing*, 14(2):265–282.

Quint, P. and Kratzke, N. (2018). Towards a lightweight multi-cloud DSL for elastic and transferable cloud-native applications. In *CLOSER*, pages 400–408. SciTePress.

Rodrguez, P., Haghighatkhah, A., Lwakatare, L. E., Teppola, S., Suomalainen, T., Eskeli, J., Karvonen, T., Kuvaja, P., Verner, J. M., and Oivo, M. (2017). Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, 123:263 – 291.

Saatkamp, K., Breitenbücher, U., Kopp, O., and Leymann, F. (2017). Topology splitting and matching for multi-cloud deployments. In *CLOSER*, pages 247–258.

Saltzer, J. H. and Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308.

Sippu, S. and Soisalon-Soininen, E. (1982). On ll(k) parsing. *Information and Control*, 53(3):141 – 164.

Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. (2007). Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287.

Wilder, B. (2012). *Cloud architecture patterns: using microsoft azure.* " O'Reilly Media, Inc.".

Zaytsev, V. (2012). Bnf was here: What have we done about the unnecessary diversity of notation for syntactic definitions. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1910–1915, New York, NY, USA. ACM.

Appendix A

Grammar

A.1 The Antlr4 Rules

```
grammar Velo;

/*
 * Parser Rules
 */

velo: author NL orchestration NL? EOF;
author: AUTHOR EQUAL BASIC_STRING;
orchestration: svBag | mvBag;
svBag:
        sBagHeader NL applicationId NL (instances NL)? (cmd NL)? (
                cpus NL
```

```
        )? (disk NL)? (memory NL)? (acceptedResourceRoles NL)? (
                args NL
        )? (backoffFactor NL)? (backoffSeconds NL)? (matchExpressions NL)? (
                dependencies NL
        )? (executor NL)? (maxLaunchDelay NL)? (requirePorts NL)? (
                user NL
        )? (version NL)? (taskKillGracePeriod NL)? (tty NL)? container NL envs? labels? (
                upgradeStrategy NL
        )? (secrets NL)? (healthChecks NL)* (readinessChecks NL)* portDefinitions* (
                fetch NL
        )*;
sBagHeader: OPENBR 'svBag' CLOSINGBR;
acceptedResourceRoles: ACCEPTEDROLES EQUAL str_array;
args: ARGS EQUAL str_array;
backoffSeconds: BACKOFFSECS EQUAL (DEC_INT | FLOAT);
check:
        checkTable NL delaySeconds NL exec NL intervalSeconds NL timeoutSeconds NL checkHttp;
checkTable: OPENBR 'mvBag.containers.check' CLOSINGBR;
delaySeconds: DELAYSEC EQUAL (DEC_INT | FLOAT);
intervalSeconds: INTERVALSEC EQUAL (DEC_INT | FLOAT);
timeoutSeconds: TIMEMOUTSEC EQUAL (DEC_INT | FLOAT);
checkHttp: httpTable NL path NL portIndex NL scheme NL checkTcp;
httpTable: OPENBR 'mvBag.containers.check.http' CLOSINGBR;
path: PATH EQUAL BASIC_STRING;
portIndex: PORTINDEX EQUAL (DEC_INT | FLOAT);
scheme: SCHEME EQUAL BASIC_STRING;
checkTcp: checkTcpTable NL portIndex;
checkTcpTable: OPENBR 'mvBag.containers.check.tcp' CLOSINGBR;
cmd: CMD EQUAL BASIC_STRING;
matchExpressions: MATCHEXPRESSIONS EQUAL matchExpression;
```

```
matchExpression: OPENBR matchExpressionValues? CLOSINGBR;
matchExpressionValues: (
                NL? OPENBR comment_or_nl BASIC_STRING ',' comment_or_nl EXPRESSIONOPERATOR ',' comment_or_nl
                        BASIC_STRING NL? CLOSINGBR NL? ',' matchExpressionValues
        )
        | NL? OPENBR comment_or_nl BASIC_STRING ',' comment_or_nl EXPRESSIONOPERATOR ',' comment_or_nl
                BASIC_STRING NL? CLOSINGBR NL?;
dependencies: DEPENDENCIES EQUAL str_array;
envs: envTable NL (keyValueStr NL)*;
envTable: OPENBR OPENBR 'svBag.env' CLOSINGBR CLOSINGBR;
executor: EXECUTOR EQUAL BASIC_STRING;
executorResources:
        executorResourcesTable NL cpu NL disk NL resourceMemory;
executorResourcesTable:
        OPENBR 'svBag.executorResources' CLOSINGBR;
fetch: fetchTable NL uri;
fetchTable: OPENBR OPENBR 'svBag.fetch' CLOSINGBR CLOSINGBR;
labels: labelsTable NL (keyValueStr NL)*;
labelsTable: OPENBR OPENBR 'svBag.labels' CLOSINGBR CLOSINGBR;
gpus: GPUS EQUAL (DEC_INT | FLOAT);
applicationId: ID EQUAL BASIC_STRING;
healthChecks:
        healthChecksTable NL gracePeriodSeconds NL ignoreHttp NL intervalSeconds NL
                maxConsecutiveFailures NL path NL portIndex NL protocol NL timeoutSeconds;
healthChecksTable:
        OPENBR OPENBR 'svBag.healthChecks' CLOSINGBR CLOSINGBR;
instances: INSTANCES EQUAL DEC_INT;
killSelection: KILLSELECTION EQUAL KILLTYPE;
ports: PORTS EQUAL int_array;
readinessChecks:
```

```
        readinessChecksTable NL name NL protocol NL path NL portName NL intervalSeconds NL
                timeoutSeconds NL readyStatusCodes NL preserveLastResponse;
readinessChecksTable:
        OPENBR OPENBR 'svBag.readinessChecks' CLOSINGBR CLOSINGBR;
portName: PORTNAME EQUAL BASIC_STRING;
readyStatusCodes: HTTPREADYSTATUSCODES EQUAL int_array;
preserveLastResponse: PRESERVELASTRESPONSE EQUAL BOOLEAN;
requirePorts: REQUIREPORTS EQUAL BOOLEAN;
secrets: secretsTable NL (secretTable NL source)*;
secretsTable: OPENBR OPENBR 'svBag.secrets' CLOSINGBR CLOSINGBR;
secretTable: OPENBR 'svBag.secrets.' UNQUOTED_KEY CLOSINGBR;
source: SOURCE EQUAL BASIC_STRING NL?;
user: USER EQUAL BASIC_STRING;
upgradeStrategy:
        upgradeStrategyTable NL maximumOverCapacity NL minimumHealthCapacity;
upgradeStrategyTable: OPENBR 'svBag.upgradeStrategy' CLOSINGBR;
maximumOverCapacity: MAXIMUMOVERCAPACITY EQUAL DEC_INT;
minimumHealthCapacity: MINIMUMHEALTHCAPACITY EQUAL DEC_INT;
version: VERSION EQUAL BASIC_STRING;
taskKillGracePeriod: TASKKILLGRACEPERIODSECONDS EQUAL DEC_INT;
container:
        conatinerHeader NL (credential NL)? (forcePullImage NL)? image NL (restartPolicy NL)? (network NL)?
                (privileged NL)? singleContainerLabels? parameters? (
                portMapping NL
        )* (pullConfig NL)? (volumes NL)* commands*;
conatinerHeader: OPENBR 'svBag.container' CLOSINGBR;
forcePullImage: FORCEPULLIMAGE EQUAL BOOLEAN;
singleContainerLabels:
        singleContainerLabelsTable NL (keyValueStr NL)*;
singleContainerLabelsTable:
```

```
        OPENBR OPENBR 'svBag.container.labels' CLOSINGBR CLOSINGBR;
portMapping:
        portMappingTable NL containerPort NL hostPort NL protocol NL servicePort NL name;
portMappingTable:
        OPENBR OPENBR 'svBag.container.portMappings' CLOSINGBR CLOSINGBR;
servicePort: SERVICEPORT EQUAL DEC_INT;
image: IMAGE EQUAL BASIC_STRING;
network: NETWORK EQUAL BASIC_STRING;
parameters: parametersTable NL (keyValueStr NL)*;
parametersTable:
        OPENBR OPENBR 'svBag.container.parameters' CLOSINGBR CLOSINGBR;
privileged: PRIVILEGED EQUAL BOOLEAN;
credential: credentialTable NL principal NL credentialSecret;
credentialTable: OPENBR 'svBag.container.credential' CLOSINGBR;
principal: PRINCIPAL EQUAL BASIC_STRING;
credentialSecret: SECRET EQUAL BASIC_STRING;
volumes:
        volumesTable NL containerPath NL hostPath NL volumeMode;
volumesTable:
        OPENBR OPENBR 'svBag.container.volumes' CLOSINGBR CLOSINGBR;
hostPath: HOSTPATH EQUAL BASIC_STRING;
portDefinitions:
        portDefinitionsTable NL port NL protocol NL name NL portLabels;
portDefinitionsTable:
        OPENBR OPENBR 'svBag.portDefinitions' CLOSINGBR CLOSINGBR;
port: PORT EQUAL DEC_INT;
portLabels: portLabelsTable NL (keyValueStr NL)*;
portLabelsTable:
        OPENBR OPENBR 'svBag.portDefinitions.labels' CLOSINGBR CLOSINGBR;
mvBag:
```

```
        mvBagHeader NL mvBagID NL (instances NL)? (mvBagNetworks NL)? (
                matchExpressions NL
        )? (mvBagNetwork NL)? (mvBagSecret NL)? (mvBagUser NL)? (
                mvBagVersion NL
        )? mvBagEnv? (mvBagExecutorResources NL)? mvBagLabels? (
                mvBagScaling NL
        )? (mvBagScheduling NL)? (mvBagContainers+ NL?)+ mvBagVolumes*;
mvBagHeader: OPENBR 'mvBag' CLOSINGBR;
mvBagID: applicationId;
restartPolicy: RESTART EQUAL RESTARTPOLICY;
mvBagNetworks: NETWORKS EQUAL str_array;
mvBagNetwork: mvBagNetworkTable NL keyValueStr;
mvBagNetworkTable: OPENBR 'mvBag.exposeOn' CLOSINGBR;
mvBagSecret: mvBagSecretsTable NL (mvBagSecretTable NL source)*;
mvBagSecretsTable:
        OPENBR OPENBR 'mvBag.secrets' CLOSINGBR CLOSINGBR;
mvBagSecretTable:
        OPENBR 'mvBag.secrets.' UNQUOTED_KEY CLOSINGBR;
mvBagUser: user;
mvBagVersion: version;
mvBagEnv: mvBagEnvTable NL (keyValueStr NL)*;
mvBagEnvTable:
        OPENBR OPENBR 'mvBag.environment' CLOSINGBR CLOSINGBR;
mvBagExecutorResources:
        mvBagExecutorResourcesTable NL cpu NL disk NL resourceMemory;
mvBagExecutorResourcesTable:
        OPENBR 'mvBag.executorResources' CLOSINGBR;
cpu: CPU EQUAL (FLOAT | DEC_INT);
resourceMemory: MEMORY EQUAL (FLOAT | DEC_INT);
disk: DISK EQUAL (FLOAT | DEC_INT);
```

```
mvBagLabels: mvBagLabelsTable NL (keyValueStr NL)*;
mvBagLabelsTable:
        OPENBR OPENBR 'mvBag.labels' CLOSINGBR CLOSINGBR;
mvBagScaling: mvBagScalingTable NL scalingType;
mvBagScalingTable: OPENBR 'mvBag.scaling' CLOSINGBR;
scalingType: KIND EQUAL BASIC_STRING;
mvBagScheduling:
        mvBagSchedulingTable NL killSelection NL placement NL unreachableStrategy NL upgrade NL
                schedulingBackoff;
mvBagSchedulingTable: OPENBR 'mvBag.scheduling' CLOSINGBR;
placement: PLACEMENT EQUAL BASIC_STRING;
unreachableStrategy: UNREACHABLESTRATEGY EQUAL BASIC_STRING;
upgrade: UPGRADE EQUAL BASIC_STRING;
schedulingBackoff:
        schedulingBackoffTable NL backoff NL backoffFactor NL maxLaunchDelay;
schedulingBackoffTable:
        OPENBR 'mvBag.scheduling.backoff' CLOSINGBR;
backoff: BACKOFF EQUAL (DEC_INT | FLOAT);
backoffFactor: BACKOFFFACTOR EQUAL (DEC_INT | FLOAT);
maxLaunchDelay: MAXLAUNCHDELAY EQUAL (DEC_INT | FLOAT);
mvBagContainers:
        containerTable NL (exec NL)? name NL (tty NL)? (
                restartPolicy NL
        )? (containerUser NL)? (check NL)? containerEnv? (
                containerHealthCheck NL
        )? containerImage containerLabels? (containerLifeCycle NL)? (
                containerResources NL
        )? (containerArtifacts NL)* containerNetworks* containerVolumes*;
containerTable:
        OPENBR OPENBR 'mvBag.containers' CLOSINGBR CLOSINGBR;
```

```
exec: EXEC EQUAL BASIC_STRING;
name: NAME EQUAL BASIC_STRING;
tty: TTY EQUAL BOOLEAN;
containerUser: mvBagUser;
containerEnv: containerEnvTable NL (keyValueStr NL)*;
containerEnvTable:
        OPENBR OPENBR 'mvBag.containers.environment' CLOSINGBR CLOSINGBR;
containerHealthCheck:
        containerHealthCheckTable NL (gracePeriodSeconds NL)? (
                ignoreHttp NL
        )? (intervalSeconds NL)? maxConsecutiveFailures NL (path NL)? (
                portIndex NL
        )? protocol NL timeoutSeconds;
containerHealthCheckTable:
        OPENBR 'mvBag.containers.healthcheck' CLOSINGBR;
gracePeriodSeconds: GRACEPER EQUAL (DEC_INT | FLOAT);
ignoreHttp: INGNOREHTTP EQUAL BOOLEAN;
maxConsecutiveFailures: MAXFAILURE EQUAL (DEC_INT | FLOAT);
protocol: PROTOCOL EQUAL BASIC_STRING;
containerImage:
        imageTable NL forcePull NL imageID NL imageKind NL (
                pullConfig NL
        )? imageLabels? imageCommands*;
imageTable: OPENBR 'mvBag.containers.image' CLOSINGBR;
forcePull: FORCEPULL EQUAL BOOLEAN;
imageID: applicationId;
imageKind: scalingType;
pullConfig: PULLCONFIG EQUAL BASIC_STRING;
imageLabels: imageLabelsTable NL (keyValueStr NL)*;
imageLabelsTable:
```

```
        OPENBR OPENBR 'mvBag.containers.image.labels' CLOSINGBR CLOSINGBR;
containerLabels: containerLabelsTable NL (keyValueStr NL)*;
containerLabelsTable:
        OPENBR OPENBR 'mvBag.containers.labels' CLOSINGBR CLOSINGBR;
containerLifeCycle: containerLifeCycleTable NL killGracePeriod;
containerLifeCycleTable:
        OPENBR 'mvBag.containers.lifecycle' CLOSINGBR;
killGracePeriod: KILLGRACEPERIODSECONDS EQUAL DEC_INT;
containerResources: containerResourcesTable NL cpus NL memory;
containerResourcesTable:
        OPENBR 'mvBag.containers.resources' CLOSINGBR;
cpus: CPUS EQUAL DEC_INT;
memory: MEM EQUAL DEC_INT;
containerArtifacts: containerArtifactsTable NL uri;
containerArtifactsTable:
        OPENBR OPENBR 'mvBag.containers.artifacts' CLOSINGBR CLOSINGBR;
uri: URI EQUAL BASIC_STRING;
containerNetworks:
        containerNetworkTable NL containerPort NL hostPort NL name NL (
                networkNames NL
        )? (protocol NL)? networkLabels?;
containerNetworkTable:
        OPENBR OPENBR 'mvBag.containers.networks' CLOSINGBR CLOSINGBR;
containerPort: CONTAINERPORT EQUAL DEC_INT;
hostPort: HOSTPORT EQUAL DEC_INT;
networkNames: NETWORKNAMES EQUAL str_array;
str_array: OPENBR str_array_values? comment_or_nl CLOSINGBR;
str_array_values: (
                comment_or_nl BASIC_STRING ',' str_array_values comment_or_nl
        )
```

```
        | comment_or_nl BASIC_STRING ','?;
int_array: '[' int_array_values? comment_or_nl ']';
int_array_values: (
                comment_or_nl DEC_INT ',' int_array_values comment_or_nl
        )
        | comment_or_nl DEC_INT ','?;
networkLabels: networkLabelsTable NL (keyValueStr NL)*;
networkLabelsTable:
        OPENBR OPENBR 'mvBag.containers.networks.labels' CLOSINGBR CLOSINGBR;
containerVolumes:
        containerVolumesTable NL mountPath NL name NL readOnly;
containerVolumesTable:
        OPENBR OPENBR 'mvBag.containers.volumeMounts' CLOSINGBR CLOSINGBR;
mountPath: MOUNTPATH EQUAL BASIC_STRING;
readOnly: READONLY EQUAL BOOLEAN;
mvBagVolumes:
        mvBagVolumesTable NL containerPath NL volumeMode NL volumePersistance;
mvBagVolumesTable:
        OPENBR OPENBR 'mvBag.volumes' CLOSINGBR CLOSINGBR;
containerPath: CONTAINERPATH EQUAL BASIC_STRING;
volumeMode: MODE EQUAL BASIC_STRING;
volumePersistance:
        volumePersistanceTable NL maxSize NL profileName NL size NL persistanceType NL constraint;
volumePersistanceTable:
        OPENBR 'mvBag.volumes.persistent' CLOSINGBR;
maxSize: MAXSIZE EQUAL DEC_INT;
profileName: PROFILENAME EQUAL BASIC_STRING;
size: SIZE EQUAL DEC_INT;
persistanceType: TYPE EQUAL BASIC_STRING;
constraint: constraintTable NL minItems NL maxItems;
```

```
constraintTable:
        OPENBR OPENBR 'mvBag.volumes.persistent.constraints' CLOSINGBR CLOSINGBR;
minItems: MINITEMS EQUAL DEC_INT;
maxItems: MAXITEMS EQUAL DEC_INT;
comment_or_nl: (COMMENT? NL)*;
keyValueStr: key '=' BASIC_STRING;
key: simpleKey | dottedKey;
simpleKey: UNQUOTED_KEY;
dottedKey: simpleKey ('.' simpleKey)+;
imageCommands:
        imageCommandsTable comment_or_nl (workDir comment_or_nl)? (
                runShell comment_or_nl
        )? (cmdShell comment_or_nl)? (runExec comment_or_nl)? (
                cmdExec comment_or_nl
        )? (copyCommand comment_or_nl)? (entrypoint comment_or_nl)? (
                addCommand comment_or_nl
        )?;
imageCommandsTable:
        OPENBR OPENBR 'mvBag.containers.image.commands' CLOSINGBR CLOSINGBR;
commands:
        commandsTable comment_or_nl (workDir comment_or_nl)? (
                runShell comment_or_nl
        )? (cmdShell comment_or_nl)? (runExec comment_or_nl)? (
                cmdExec comment_or_nl
        )? (copyCommand comment_or_nl)? (entrypoint comment_or_nl)? (
                addCommand comment_or_nl
        )?;

commandsTable:
        OPENBR OPENBR 'svBag.container.commands' CLOSINGBR CLOSINGBR;
```

```
runExec: RUN EQUAL BASIC_STRING;
runShell: RUNSHELL EQUAL str_array;
cmdExec: CMD EQUAL BASIC_STRING;
cmdShell: CMDSHELL EQUAL str_array;
addCommand:
        ADD EQUAL OPENBR NL? src ',' NL? BASIC_STRING NL? CLOSINGBR;
src: '.' | BASIC_STRING '.' BASIC_STRING | BASIC_STRING;
copyCommand:
        COPY EQUAL OPENBR NL? src ',' NL? BASIC_STRING NL? CLOSINGBR;
workDir: WORKDIR EQUAL src;
entrypoint:
        ENTRYPOINT EQUAL OPENBR NL? BASIC_STRING ',' NL? BASIC_STRING NL? CLOSINGBR;
/*
 * Lexer Rules
 */
ENTRYPOINT: 'entrypoint';
WORKDIR: 'workdir';
ADD: 'add';
COPY: 'copy';
RUN: 'run';
RUNSHELL: 'runShell';
ARGS: 'args';
MATCHEXPRESSIONS: 'matchExpressions';
DEPENDSON: 'dependsOn';
ACCEPTEDROLES: 'acceptedResourceRoles';
PORTS: 'ports';
INSTANCES: 'instances';
CMD: 'cmd';
CMDSHELL: 'cmdShell';
EXECUTOR: 'executor';
```

```
RESTARTPOLICY:
        '"' 'Always' '"'
        | '"' 'OnFailure' '"'
        | '"' 'Never' '"';
RESTART: 'restart';
EXPRESSIONOPERATOR:
        '"' 'UNIQUE' '"'
        | '"' 'CLUSTER' '"'
        | '"' 'GROUP_BY' '"'
        | '"' 'LIKE' '"'
        | '"' 'UNLIKE' '"'
        | '"' 'MAX_PER' '"'
        | '"' 'IS' '"'
        | '"' 'In' '"'
        | '"' 'NotIn' '"'
        | '"' 'Exists' '"'
        | '"' 'DoesNotExist' '"';
DEPENDENCIES: 'dependencies';
AUTHOR: 'author';
KILLSELECTION: 'killSelection';
PLACEMENT: 'placement';
UNREACHABLESTRATEGY: 'unreachableStrategy';
UPGRADE: 'upgrade';
EQUAL: '=';
KIND: 'kind';
ID: 'id';
IMAGE: 'image';
SECRETS: 'secrets';
SECRET: 'secret';
USER: 'user';
```

```
VERSION: 'version';
BACKOFF: 'backoff';
BACKOFFFACTOR: 'backoffFactor';
BACKOFFSECS: 'backoffSeconds';
MAXLAUNCHDELAY: 'maxLaunchDelaySeconds';
EXEC: 'exec';
NAME: 'name';
TTY: 'tty';
DELAYSEC: 'delaySeconds';
INTERVALSEC: 'intervalSeconds';
TIMEMOUTSEC: 'timeoutSeconds';
PATH: 'path';
PORTINDEX: 'portIndex';
SCHEME: 'scheme';
GRACEPER: 'gracePeriodSeconds';
INGNOREHTTP: 'ignoreHttp1xx';
MAXFAILURE: 'maxConsecutiveFailures';
PROTOCOL: 'protocol';
FORCEPULL: 'forcePull';
FORCEPULLIMAGE: 'forcePullImage';
PULLCONFIG: 'pullConfig';
KILLGRACEPERIODSECONDS: 'killGracePeriodSeconds';
TASKKILLGRACEPERIODSECONDS: 'taskKillGracePeriodSeconds';
KILLTYPE: '"' 'YOUNGEST_FIRST' '"' | '"' 'OLDEST_FIRST' '"';
CPUS: 'cpus';
MEM: 'mem';
CPU: 'cpu';
GPUS: 'gpus';
MEMORY: 'memory';
DISK: 'disk';
```

```
URI: 'uri';
CONTAINERPORT: 'containerPort';
HOSTPORT: 'hostPort';
HOSTPATH: 'hostPath';
NETWORKNAMES: 'networkNames';
NETWORKS: 'networks';
NETWORK: 'network';
MOUNTPATH: 'mountPath';
READONLY: 'readOnly';
CONTAINERPATH: 'containerPath';
MODE: 'mode';
MAXSIZE: 'maxSize';
PROFILENAME: 'profileName';
SIZE: 'size';
MINITEMS: 'minItems';
MAXITEMS: 'maxItems';
TYPE: 'type';
OPENBR: '[';
CLOSINGBR: ']';
PORT: 'port';
SERVICEPORT: 'servicePort';
PORTNAME: 'portName';
HTTPREADYSTATUSCODES: 'httpStatusCodesForReady';
PRESERVELASTRESPONSE: 'preserveLastResponse';
REQUIREPORTS: 'requirePorts';
MAXIMUMOVERCAPACITY: 'maximumOverCapacity';
MINIMUMHEALTHCAPACITY: 'minimumHealthCapacity';
SOURCE: 'source';
PRIVILEGED: 'privileged';
PRINCIPAL: 'principal';
```

```
WS: [ \t]+ -> skip;
NL: ('\r'? '\n')+;
COMMENT: '#' (~[\n])*;

fragment DIGIT: [0-9];
fragment ALPHA: [A-Za-z];

// booleans
BOOLEAN: 'true' | 'false';

// strings
fragment ESC: '\\' (["\\/bfnrt] | UNICODE | EX_UNICODE);
fragment ML_ESC: '\\' '\r'? '\n' | ESC;
fragment UNICODE: 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT;
fragment EX_UNICODE:
        'U' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT;
BASIC_STRING: '"' (ESC | ~["\\\n])*? '"';
ML_BASIC_STRING: '"""' (ML_ESC | ~["\\])*? '"""';
LITERAL_STRING: '\'' (~['\n])*? '\'';
ML_LITERAL_STRING: '\'\'\'' (.)*? '\'\'\'';
// floating point numbers
fragment EXP: ('e' | 'E') DEC_INT;
fragment ZERO_PREFIXABLE_INT: DIGIT (DIGIT | '_' DIGIT)*;
fragment FRAC: '.' ZERO_PREFIXABLE_INT;
FLOAT: DEC_INT ( EXP | FRAC EXP?);
INF: [+-]? 'inf';
NAN: [+-]? 'nan';
// integers
fragment HEX_DIGIT: [A-F] | DIGIT;
```

```
fragment DIGIT_1_9: [1-9];
fragment DIGIT_0_7: [0-7];
fragment DIGIT_0_1: [0-1];
DEC_INT: [+-]? (DIGIT | (DIGIT_1_9 (DIGIT | '_' DIGIT)+));
HEX_INT: '0x' HEX_DIGIT (HEX_DIGIT | '_' HEX_DIGIT)*;
OCT_INT: '0o' DIGIT_0_7 (DIGIT_0_7 | '_' DIGIT_0_7);
BIN_INT: '0b' DIGIT_0_1 (DIGIT_0_1 | '_' DIGIT_0_1)*;
UNQUOTED_KEY: (ALPHA | DIGIT | '-' | '_')+;
```

Appendix B

Specifications

B.1 Specifications of Application in section 5.1

```
author = "John Doe john@example.com"
    [mvBag]
    id = "bookingwebapp"
    instances = 2
    networks = ["Net1","Net2"]
    matchExpressions = [["tier","In","cache"],[ "environment","NotIn","dev"]]
    user = "gervasius"
    version = "20190601"

        [[mvBag.labels]]
        App = "bookingwebapp"
        Environment ="production"

        [[mvBag.containers]]
        name = "pythonapp"
        restart = "Always"
        user = "admin"

            [mvBag.containers.healthcheck]
            gracePeriodSeconds = 30.0
            ignoreHttp1xx = false
            intervalSeconds = 30.0
            maxConsecutiveFailures = 3.0
            path = "somePath"
            portIndex = 1.0
            protocol = "TCP"
            timeoutSeconds = 60.0

            [mvBag.containers.image]
            forcePull = true
            id = "ubuntu:16.04"
            kind = "Docker"

                [[mvBag.containers.image.labels]]
                lastUpdateBy = "root"

                [[mvBag.containers.image.commands]]
                #install git
                Rochelle = ["install","y", "git core"]
                #install python
                run = ["install",python-pip"]

               [[mvBag.containers.image.commands]]
                #pull the python app
                runShell = ["git","clone", "https://git.logicpp.net/app/python-app.git"]
               #run the python ap
               run = "python app/python-app.py"

           [[mvBag.containers.labels]]
           tier = "front-end"
            App = "booking-web-app"

           [mvBag.containers.resources]
            cpus = 4
```

```
55              mem = 32
56
57              [[mvBag.containers.volumeMounts]]
58              mountPath = "/docker_storage/booking"
59              name = "bookingFolder"
60              readOnly = false
61
62          [[mvBag.containers]]
63          name = "ngnix"
64          restart = "OnFailure"
65          user = "root"
66
67              [mvBag.containers.image]
68              forcePull = true
69              id = "nginx:1.15.8-alpine"
70              kind = "Docker"
71
72                  [[mvBag.containers.image.commands]]
73                  workdir = "/app"
74                  #Remove default Nginx website
75                  run = "rm -rf /usr/share/nginx/html/*"
76                    copy = ["./dev/nginx.conf","/etc/nginx/nginx.conf"]
77                    entrypoint = ["sh", "run.sh"]
78
79                 [[mvBag.containers.labels]]
80                 tier = "front-end"
81                 App = "booking-web-app"
82
83                 [[mvBag.containers.networks]]
84                 containerPort = 80
85                 hostPort = 80
86                 name = "frontEndPort"
87                 protocol = "TCP"
```

Listing B.1: Web application specification